EBook Gratis

APRENDIZAJE

Scala Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#scala

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: scala-language

It is an unofficial and free Scala Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Scala Language.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con Scala Language

Observaciones

Scala es un moderno lenguaje de programación multi-paradigma diseñado para expresar patrones de programación comunes de una manera concisa, elegante y segura. Se integra sin problemas las características de los lenguajes orientados a objetos y funcionales .

La mayoría de los ejemplos dados requieren una instalación de Scala en funcionamiento. Esta es la página de instalación de Scala , y este es el ejemplo 'Cómo configurar Scala' . scalafiddle.net es un buen recurso para ejecutar pequeños ejemplos de código en la web.

Versiones

Versión	Fecha de lanzamiento
2.10.1	2013-03-13
2.10.2	2013-06-06
2.10.3	2013-10-01
2.10.4	2014-03-24
2.10.5	2015-03-05
2.10.6	2015-09-18
2.11.0	2014-04-21
2.11.1	2014-05-21
2.11.2	2014-07-24
2.11.4	2014-10-30
2.11.5	2014-01-14
2.11.6	2015-03-05
2.11.7	2015-06-23
2.11.8	2016-03-08
2.11.11	2017-04-19
2.12.0	2016-11-03

Versión	Fecha de lanzamiento
2.12.1	2016-12-06
2.12.2	2017-04-19

Examples

Hola mundo definiendo un método 'principal'

Coloque este código en un archivo llamado `HelloWorld.scala` :

```
object Hello {
  def main(args: Array[String]): Unit = {
    println("Hello World!")
  }
}
```

Demo en vivo

Para compilarlo a bytecode que es ejecutable por la JVM:

```
$ scalac HelloWorld.scala
```

Para ejecutarlo:

```
$ scala Hello
```

Cuando el tiempo de ejecución de Scala carga el programa, busca un objeto llamado `Hello` con un método `main` . El método `main` es el punto de entrada del programa y se ejecuta.

Tenga en cuenta que, a diferencia de Java, Scala no tiene el requisito de nombrar objetos o clases después del archivo en el que se encuentran. En su lugar, el parámetro `Hello` pasó en el comando `scala Hello` refiere al objeto que se busca que contiene el método `main` que se ejecutará. Es perfectamente posible tener varios objetos con métodos principales en el mismo archivo `.scala` .

La matriz `args` contendrá los argumentos de línea de comando dados al programa, si los hay. Por ejemplo, podemos modificar el programa así:

```
object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Hello World!")
    for {
      arg <- args
    } println(s"Arg=$arg")
  }
}
```

Compilarlo

```
$ scalac HelloWorld.scala
```

Y luego ejecutarlo:

```
$ scala HelloWorld 1 2 3
Hello World!
Arg=1
Arg=2
Arg=3
```

Hola mundo extendiendo la aplicación

```
object HelloWorld extends App {
  println("Hello, world!")
}
```

Demo en vivo

Al extender el rasgo de la `App` , puede evitar definir un método `main` explícito. Todo el cuerpo del objeto `HelloWorld` se trata como "el método principal".

2.11.0

> ## Inicialización retrasada
>
> Según la documentación oficial , la `App` hace uso de una función llamada *Inicialización diferida* . Esto significa que los campos de objeto se inicializan *después de* llamar al método principal.

2.11.0

> ## Inicialización retrasada
>
> Según la documentación oficial , la `App` hace uso de una función llamada *Inicialización diferida* . Esto significa que los campos de objeto se inicializan *después de* llamar al método principal.
>
> `DelayedInit` ahora está en **desuso** para uso general, pero *todavía* es *compatible* con la `App` como un caso especial. El soporte continuará hasta que se decida e implemente una función de reemplazo.

Para acceder a los argumentos de la línea de comandos al extender la `App` , use `this.args` :

```
object HelloWorld extends App {
  println("Hello World!")
  for {
    arg <- this.args
  } println(s"Arg=$arg")
}
```

Al usar la `App` , el cuerpo del objeto se ejecutará como el método `main` , no hay necesidad de anular `main` .

Hola mundo como guión

Scala puede ser usado como un lenguaje de scripting. Para demostrarlo, cree `HelloWorld.scala` con el siguiente contenido:

```
println("Hello")
```

Ejecútelo con el intérprete de línea de comandos (el `$` es el indicador de línea de comandos):

```
$ scala HelloWorld.scala
Hello
```

Si omite `.scala` (como si simplemente escribió `scala HelloWorld`) el corredor buscará un archivo `.class` compilado con `.scala` lugar de compilar y luego ejecutar el script.

> **Nota:** Si se usa scala como lenguaje de scripting, no se puede definir ningún paquete.

En los sistemas operativos que utilizan `bash` o terminales de shell similares, los scripts de Scala se pueden ejecutar utilizando un 'preámbulo de shell'. Cree un archivo llamado `HelloWorld.sh` y coloque lo siguiente como contenido:

```
#!/bin/sh
exec scala "$0" "$@"
!#
println("Hello")
```

Las partes entre `#!` y `!#` es el 'preámbulo de shell', y se interpreta como un script bash. El resto es Scala.

Una vez que haya guardado el archivo anterior, debe otorgarle permisos 'ejecutables'. En la cáscara puedes hacer esto:

```
$ chmod a+x HelloWorld.sh
```

(Tenga en cuenta que esto le da permiso a todos: lea acerca de chmod para aprender cómo configurarlo para conjuntos de usuarios más específicos).

Ahora puedes ejecutar el script así:

```
$ ./HelloWorld.sh
```

Usando el Scala REPL

Cuando ejecuta `scala` en un terminal sin parámetros adicionales, se abre un intérprete REPL (Read-Eval-Print Loop):

```
nford:~ $ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_66).
Type in expressions for evaluation. Or try :help.

scala>
```

El REPL le permite ejecutar Scala en forma de hoja de trabajo: el contexto de ejecución se conserva y puede probar los comandos manualmente sin tener que construir un programa completo. Por ejemplo, al escribir `val poem = "As halcyons we shall be"` se vería así:

```
scala> val poem = "As halcyons we shall be"
poem: String = As halcyons we shall be
```

Ahora podemos imprimir nuestro `val` :

```
scala> print(poem)
As halcyons we shall be
```

Tenga en cuenta que `val` es inmutable y no se puede sobrescribir:

```
scala> poem = "Brooding on the open sea"
<console>:12: error: reassignment to val
       poem = "Brooding on the open sea"
```

Pero en el REPL *puede* redefinir un valor `val` (lo que causaría un error en un programa Scala normal, si se realizó en el mismo ámbito):

```
scala> val poem = "Brooding on the open sea"
poem: String = Brooding on the open sea
```

Para el resto de su sesión REPL, esta variable recién definida sombreará la variable previamente definida. Los REPL son útiles para ver rápidamente cómo funcionan los objetos u otros códigos. Todas las funciones de Scala están disponibles: puede definir funciones, clases, métodos, etc.

Scala Quicksheet

Descripción	Código
Asignar valor int inmutable	`val x = 3`
Asignar valor int mutable	`var x = 3`
Asignar valor inmutable con tipo explícito	`val x: Int = 27`
Asignar valor perezosamente evaluado	`lazy val y = print("Sleeping in.")`
Vincular una función a un nombre	`val f = (x: Int) => x * x`
Enlazar una función a un nombre con tipo explícito	`val f: Int => Int = (x: Int) => x * x`

Descripción	Código
Define un método	`def f(x: Int) = x * x`
Definir un método con tipificación explícita.	`def f(x: Int): Int = x * x`
Define una clase	`class Hopper(someParam: Int) { ... }`
Define un objeto	`object Hopper(someParam: Int) { ... }`
Definir un rasgo	`trait Grace { ... }`
Obtener primer elemento de secuencia	`Seq(1,2,3).head`
Si interruptor	`val result = if(x > 0) "Positive!"`
Obtener todos los elementos de secuencia excepto primero	`Seq(1,2,3).tail`
Recorrer una lista	`for { x <- Seq(1,2,3) } print(x)`
Bucle anidado	`for { x <- Seq(1,2,3) y <- Seq(4,5,6) } print(x + ":" + y)`
Para cada elemento de lista ejecutar la función.	`List(1,2,3).foreach { println }`
Imprimir a estándar	`print("Ada Lovelace")`
Ordenar una lista alfanuméricamente	`List('b','c','a').sorted`

Lea Empezando con Scala Language en línea:
https://riptutorial.com/es/scala/topic/216/empezando-con-scala-language

Capítulo 2: Alcance

Introducción

El alcance en Scala define desde dónde se puede acceder a un valor (`def` , `val` , `var` o `class`).

Sintaxis

- declaración
- declaración privada
- Declaración privada [este]
- Declaración privada de [fromWhere]
- declaración protegida
- Declaración protegida [de donde]

Examples

Ámbito público (predeterminado)

Por defecto, el alcance es `public` , se puede acceder al valor desde cualquier lugar.

```
package com.example {
  class FooClass {
    val x = "foo"
  }
}

package an.other.package {
  class BarClass {
    val foo = new com.example.FooClass
    foo.x // <- Accessing a public value from another package
  }
}
```

Un ámbito privado

Cuando el alcance es privado, solo se puede acceder desde la clase actual u otras instancias de la clase actual.

```
package com.example {
  class FooClass {
    private val x = "foo"
    def aFoo(otherFoo: FooClass) {
      otherFoo.x // <- Accessing from another instance of the same class
    }
  }
  class BarClass {
    val f = new FooClass
```

```
    f.x // <- This will not compile
  }
}
```

Un paquete privado de alcance específico.

Puede especificar un paquete donde se puede acceder al valor privado.

```
package com.example {
  class FooClass {
    private val x = "foo"
    private[example] val y = "bar"
  }
  class BarClass {
    val f = new FooClass
    f.x // <- Will not compile
    f.y // <- Will compile
  }
}
```

Objeto de ámbito privado

El ámbito más restrictivo es el ámbito *"objeto-privado"* , que solo permite acceder a ese valor desde la misma instancia del objeto.

```
class FooClass {
  private[this] val x = "foo"
  def aFoo(otherFoo: FooClass) = {
    otherFoo.x // <- This will not compile, accessing x outside the object instance
  }
}
```

Alcance protegido

El ámbito protegido permite acceder al valor desde cualquier subclase de la clase actual.

```
class FooClass {
  protected val x = "foo"
}
class BarClass extends FooClass {
  val y = x // It is a subclass instance, will compile
}
class ClassB {
  val f = new FooClass
  f.x // <- This will not compile
}
```

Paquete protegido alcance

El alcance del paquete protegido permite acceder al valor solo desde cualquier subclase en un paquete específico.

```
package com.example {
  class FooClass {
    protected[example] val x = "foo"
  }
  class ClassB extends FooClass {
    val y = x // It's in the protected scope, will compile
  }
}
package com {
  class BarClass extends com.example.FooClass {
    val y = x // <- Outside the protected scope, will not compile
  }
}
```

Lea Alcance en línea: https://riptutorial.com/es/scala/topic/9705/alcance

Capítulo 3: Anotaciones

Sintaxis

- @AnAnnotation def someMethod = {...}
- @AnAnnotation class someClass {...}
- @AnnotatioWithArgs (annotation_args) def someMethod = {...}

Parámetros

Parámetro	Detalles
@	Indica que el token siguiente es una anotación.
Alguna anotación	El nombre de la anotación.
constructor_args	(Opcional) Los argumentos pasados a la anotación. Si no hay ninguno, los paréntesis son innecesarios.

Observaciones

Scala-lang proporciona una lista de anotaciones estándar y sus equivalentes de Java .

Examples

Usando una anotación

Esta anotación de muestra indica que el siguiente método está en `deprecated` .

```
@deprecated
def anUnusedLegacyMethod(someArg: Any) =  {
  ...
}
```

Esto también se puede escribir de manera equivalente como:

```
@deprecated def anUnusedLegacyMethod(someArg: Any) =  {
  ...
}
```

Anotando el constructor principal.

```
/**
```

```
 * @param num Numerator
 * @param denom Denominator
 * @throws ArithmeticException in case `denom` is `0`
 */
class Division @throws[ArithmeticException](/*no annotation parameters*/) protected (num: Int,
denom: Int) {
    private[this] val wrongValue = num / denom

    /** Integer number
     *  @param num Value */
    protected[Division] def this(num: Int) {
      this(num, 1)
    }
}
object Division {
  def apply(num: Int) = new Division(num)
  def apply(num: Int, denom: Int) = new Division(num, denom)
}
```

El modificador de visibilidad (en este caso `protected`) debe aparecer después de las anotaciones en la misma línea. En caso de que la anotación acepte parámetros opcionales (como en este caso `@throws` acepta una causa opcional), debe especificar una lista de parámetros vacía para la anotación: `()` antes de los parámetros del constructor.

Nota: Se pueden especificar múltiples anotaciones, incluso del mismo tipo (repetición de anotaciones).

De manera similar, con una clase de caso sin método de fábrica auxiliar (y causa especificada para la anotación):

```
case class Division @throws[ArithmeticException]("denom is 0") (num: Int, denom: Int) {
    private[this] val wrongValue = num / denom
}
```

Creando tus propias anotaciones

Puede crear sus propias anotaciones de Scala creando clases derivadas de scala.annotation.StaticAnnotation o scala.annotation.ClassfileAnnotation

```
package animals
// Create Annotation `Mammal`
class Mammal(indigenous:String) extends scala.annotation.StaticAnnotation

// Annotate class Platypus as a `Mammal`
@Mammal(indigenous = "North America")
class Platypus{}
```

Las anotaciones pueden ser interrogadas usando la API de reflexión.

```
scala>import scala.reflect.runtime.{universe ⇒ u}

scala>val platypusType = u.typeOf[Platypus]
platypusType: reflect.runtime.universe.Type = animals.reflection.Platypus
```

```
scala>val platypusSymbol = platypusType.typeSymbol.asClass
platypusSymbol: reflect.runtime.universe.ClassSymbol = class Platypus

scala>platypusSymbol.annotations
List[reflect.runtime.universe.Annotation] = List(animals.reflection.Mammal("North America"))
```

Lea Anotaciones en línea: https://riptutorial.com/es/scala/topic/3783/anotaciones

Capítulo 4: Auto tipos

Sintaxis

- Tipo de rasgo {selfId => / *otros miembros pueden referirse a* `selfId` *en caso de que* `this` *signifique algo* /}
- Tipo de rasgo {selfId: OtherType => / * otros miembros pueden usar `selfId` y será del tipo `OtherType` * /
- Tipo de rasgo {selfId: OtherType1 with OtherType2 => / * `selfId` es de tipo `OtherType1` y `OtherType2` * /

Observaciones

A menudo se utiliza con el patrón de pastel.

Examples

Ejemplo de auto tipo simple

Los auto tipos se pueden usar en rasgos y clases para definir restricciones en las clases concretas a las que se mezcla. También es posible utilizar un identificador diferente para la `this` usando esta sintaxis (útil cuando el objeto externo tiene que ser referenciado a partir de un objeto interno).

Supongamos que desea almacenar algunos objetos. Para eso, crea interfaces para el almacenamiento y para agregar valores a un contenedor:

```
trait Container[+T] {
  def add(o: T): Unit
}

trait PermanentStorage[T] {
  /* Constraint on self type: it should be Container
   * we can refer to that type as `identifier`, usually `this` or `self`
   * or the type's name is used. */
  identifier: Container[T] =>

  def save(o: T): Unit = {
    identifier.add(o)
    //Do something to persist too.
  }
}
```

De esta manera, no están en la misma jerarquía de objetos, pero `PermanentStorage` no se puede implementar sin implementar también `Container` .

Lea Auto tipos en línea: https://riptutorial.com/es/scala/topic/4639/auto-tipos

Capítulo 5: Biblioteca de continuaciones

Introducción

El estilo de paso de continuación es una forma de flujo de control que implica pasar a las funciones el resto de la computación como un argumento de "continuación". La función en cuestión luego invoca esa continuación para continuar la ejecución del programa. Una forma de pensar en una continuación es como un cierre. La biblioteca de continuaciones de Scala trae continuaciones delimitadas en la forma de los `shift` primitivos / `reset` al lenguaje.

biblioteca de continuaciones: https://github.com/scala/scala-continuations

Sintaxis

- reset {...} // Las continuaciones se extienden hasta el final del bloque de reinicio adjunto
- shift {...} // Crear una continuación indicando después de la llamada, pasándola al cierre
- A @cpsParam [B, C] // Un cálculo que requiere una función A => B para crear un valor de C
- @cps [A] // Alias para @cpsParam [A, A]
- @suspendable // Alias para @cpsParam [Unidad, Unidad]

Observaciones

`shift` y `reset` son estructuras de flujo de control primitivas, como `Int.+` es una operación primitiva y `Long` es un tipo primitivo. Son más primitivos que cualquiera de los dos en que las continuaciones delimitadas se pueden usar para construir casi todas las estructuras de flujo de control. No son muy útiles "listos para usar", pero realmente brillan cuando se usan en bibliotecas para crear API ricas.

Las continuaciones y las mónadas también están estrechamente vinculadas. Las continuaciones se pueden hacer en la mónada de continuación , y las mónadas son continuaciones porque su operación de `flatMap` toma una continuación como parámetro.

Examples

Las devoluciones de llamada son continuaciones

```
// Takes a callback and executes it with the read value
def readFile(path: String)(callback: Try[String] => Unit): Unit = ???

readFile(path) { _.flatMap { file1 =>
  readFile(path2) { _.foreach { file2 =>
    processFiles(file1, file2)
  }}
}}
```

El argumento de la función para `readFile` es una continuación, en que `readFile` invoca para continuar la ejecución del programa después de que haya realizado su trabajo.

Para controlar lo que puede convertirse fácilmente en un infierno de devolución de llamada, usamos la biblioteca de continuaciones.

```
reset { // Reset is a delimiter for continuations.
  for { // Since the callback hell is relegated to continuation library machinery.
        // a for-comprehension can be used
    file1 <- shift(readFile(path1)) // shift has type (((A => B) => C) => A)
    // We use it as (((Try[String] => Unit) => Unit) => Try[String])
    // It takes all the code that occurs after it is called, up to the end of reset, and
    // makes it into a closure of type (A => B).
    // The reason this works is that shift is actually faking its return type.
    // It only pretends to return A.
    // It actually passes that closure into its function parameter (readFile(path1) here),
    // And that function calls what it thinks is a normal callback with an A.
    // And through compiler magic shift "injects" that A into its own callsite.
    // So if readFile calls its callback with parameter Success("OK"),
    // the shift is replaced with that value and the code is executed until the end of reset,
    // and the return value of that is what the callback in readFile returns.
    // If readFile called its callback twice, then the shift would run this code twice too.
    // Since readFile returns Unit though, the type of the entire reset expression is Unit
    //
    // Think of shift as shifting all the code after it into a closure,
    // and reset as resetting all those shifts and ending the closures.
    file2 <- shift(readFile(path2))
  } processFiles(file1, file2)
}

// After compilation, shift and reset are transformed back into closures
// The for comprehension first desugars to:
reset {
  shift(readFile(path1)).flatMap { file1 => shift(readFile(path2)).foreach { file2 =>
processFiles(file1, file2) } }
}
// And then the callbacks are restored via CPS transformation
readFile(path1) { _.flatMap { file1 => // We see how shift moves the code after it into a
closure
  readFile(path2) { _.foreach { file2 =>
    processFiles(file1, file2)
  }}
}}  // And we see how reset closes all those closures
// And it looks just like the old version!
```

Creando funciones que toman continuaciones

Si se llama `shift` fuera de un bloque de `reset` delimitador, se puede usar para crear funciones que crean continuaciones dentro de un bloque de `reset` . Es importante tener en cuenta que el tipo de `shift` no es solo `(((A => B) => C) => A)` , en realidad es `(((A => B) => C) => (A @cpsParam[B, C]))` . Esa anotación marca dónde se necesitan las transformaciones de CPS. Las funciones que llaman a `shift` sin `reset` tienen su tipo de retorno "infectado" con esa anotación.

Dentro de un bloque de `reset` , un valor de `A @cpsParam[B, C]` parece tener un valor de `A` , aunque en realidad es solo una simulación. La continuación que se necesita para completar el cálculo tiene el tipo `A => B` , por lo que el código que sigue un método que devuelve este tipo debe

devolver `B C` es el tipo de retorno "real" y, después de la transformación de CPS, la llamada a la función tiene el tipo `C`

Ahora, el ejemplo, tomado del Scaladoc de la biblioteca.

```
val sessions = new HashMap[UUID, Int=>Unit]
def ask(prompt: String): Int @suspendable = // alias for @cpsParam[Unit, Unit]. @cps[Unit] is
also an alias. (@cps[A] = @cpsParam[A,A])
  shift {
    k: (Int => Unit) => {
      println(prompt)
      val id = uuidGen
      sessions += id -> k
    }
  }

def go(): Unit = reset {
  println("Welcome!")
  val first = ask("Please give me a number") // Uses CPS just like shift
  val second = ask("Please enter another number")
  printf("The sum of your numbers is: %d\n", first + second)
}
```

Aquí, `ask` almacenará la continuación en un mapa, y luego otro código puede recuperar esa "sesión" y pasar el resultado de la consulta al usuario. De esta manera, `go` puede estar utilizando una biblioteca asíncrona mientras su código parece un código imperativo normal.

Lea Biblioteca de continuaciones en línea: https://riptutorial.com/es/scala/topic/8312/biblioteca-de-continuaciones

Capítulo 6: Clase de opción

Sintaxis

- clase Algunos [+ T] (valor: T) extiende la opción [T]
- objeto Ninguno extiende la Opción [Nada]
- Opción [T] (valor: T)

  El constructor creará un `Some(value)` o `None` según sea apropiado para el valor proporcionado.

Examples

Opciones como colecciones

`Option` s tienen algunas funciones útiles de orden superior que pueden entenderse fácilmente al ver las opciones como *colecciones con cero o un solo elemento* , donde `None` comporta como la colección vacía, y `Some(x)` comportan como una colección con un solo elemento, `x` .

```
val option: Option[String] = ???

option.map(_.trim) // None if option is None, Some(s.trim) if Some(s)
option.foreach(println) // prints the string if it exists, does nothing otherwise
option.forall(_.length > 4) // true if None or if Some(s) and s.length > 4
option.exists(_.length > 4) // true if Some(s) and s.length > 4
option.toList // returns an actual list
```

Usando Opción en lugar de Null

En Java (y otros lenguajes), usar `null` es una forma común de indicar que no hay un valor asociado a una variable de referencia. En Scala, se prefiere usar `Option` en lugar de usar `null` . `Option` ajusta valores que *pueden* ser `null` .

`None` es una subclase de `Option` ajusta una referencia nula. `Some` es una subclase de `Option` envuelve una referencia no nula.

Envolver una referencia es fácil:

```
val nothing = Option(null) // None
val something = Option("Aren't options cool?") // Some("Aren't options cool?")
```

Este es un código típico al llamar a una biblioteca de Java que podría devolver una referencia nula:

```
val resource = Option(JavaLib.getResource())
```

```
// if null, then resource = None
// else resource = Some(resource)
```

Si `getResource()` devuelve un valor `null` , el `resource` será un objeto `None` . De lo contrario, será un objeto `Some(resource)` . La forma preferida de manejar una `Option` es usar funciones de orden superior disponibles dentro del tipo de `Option` . Por ejemplo, si desea verificar si su valor no es `None` (similar a verificar si el `value == null`), usaría la función `isDefined` :

```
val resource: Option[Resource] = Option(JavaLib.getResource())
if (resource.isDefined) {  // resource is `Some(_)` type
  val r: Resource = resource.get
  r.connect()
}
```

De manera similar, para verificar una referencia `null` puede hacer esto:

```
val resource: Option[Resource] = Option(JavaLib.getResource())
if (resource.isEmpty) { // resource is `None` type.
  System.out.println("Resource is empty! Cannot connect.")
}
```

Es preferible que trate la ejecución condicional en el valor envuelto de una `Option` (sin usar el método 'excepcional' `Option.get`) tratando la `Option` como una mónada y usando `foreach` :

```
val resource: Option[Resource] = Option(JavaLib.getResource())
resource foreach (r => r.connect())
// if r is defined, then r.connect() is run
// if r is empty, then it does nothing
```

Si se requiere una instancia de `Resource` lugar de una instancia de `Option[Resource]`), aún puede usar la `Option` para protegerse contra valores nulos. Aquí el método `getOrElse` proporciona un valor predeterminado:

```
lazy val defaultResource = new Resource()
val resource: Resource = Option(JavaLib.getResource()).getOrElse(defaultResource)
```

El código Java no manejará fácilmente la `Option` de Scala, por lo que al pasar valores al código Java es una buena forma de desenvolver una `Option` , pasar un valor `null` o un valor predeterminado razonable cuando sea apropiado:

```
val resource: Option[Resource] = ???
JavaLib.sendResource(resource.orNull)
JavaLib.sendResource(resource.getOrElse(defaultResource)) //
```

Lo esencial

Una `Option` es una estructura de datos que contiene un solo valor o ningún valor en absoluto. Una `Option` puede considerarse como colecciones de cero o uno de los elementos.

La opción es una clase abstracta con dos hijos: `Some` y `None` .

`Some` contienen un solo valor, y `None` no contiene ningún valor.

`Option` es útil en expresiones que de lo contrario usarían `null` para representar la falta de un valor concreto. Esto protege contra una `NullPointerException` y permite la composición de muchas expresiones que pueden no devolver un valor utilizando combinadores como `Map` , `FlatMap` , etc.

Ejemplo con mapa

```
val countries = Map(
  "USA" -> "Washington",
  "UK" -> "London",
  "Germany" -> "Berlin",
  "Netherlands" -> "Amsterdam",
  "Japan" -> "Tokyo"
)

println(countries.get("USA")) // Some(Washington)
println(countries.get("France")) // None
println(countries.get("USA").get) // Washington
println(countries.get("France").get) // Error: NoSuchElementException
println(countries.get("USA").getOrElse("Nope")) // Washington
println(countries.get("France").getOrElse("Nope")) // Nope
```

`Option[A]` está **sellada** y, por lo tanto, no puede extenderse. Por lo tanto, su semántica es estable y se puede confiar en ella.

Opciones en para las comprensiones

`Option` s tienen un método `flatMap` . Esto significa que pueden ser utilizados en una comprensión. De esta manera podemos levantar funciones regulares para trabajar en las `Option` s sin tener que redefinirlas.

```
val firstOption: Option[Int] = Option(1)
val secondOption: Option[Int] = Option(2)

val myResult = for {
  firstValue <- firstOption
  secondValue <- secondOption
} yield firstValue + secondValue
// myResult: Option[Int] = Some(3)
```

Cuando uno de los valores es `None` el resultado final del cálculo también será `None` .

```
val firstOption: Option[Int] = Option(1)
val secondOption: Option[Int] = None

val myResult = for {
  firstValue <- firstOption
  secondValue <- secondOption
} yield firstValue + secondValue
// myResult: Option[Int] = None
```

Nota: este patrón se extiende más generalmente para los conceptos llamados `Monad` s. (Más información debe estar disponible en las páginas relacionadas con las comprensiones y `Monad`)

En general, no es posible mezclar diferentes mónadas en una comprensión. Pero dado que la `Option` se puede convertir fácilmente en un `Iterable` , podemos combinar fácilmente las `Option` sy `Iterable` llamando al método `.toIterable` .

```
val option: Option[Int] = Option(1)
val iterable: Iterable[Int] = Iterable(2, 3, 4, 5)

// does NOT compile since we cannot mix Monads in a for comprehension
// val myResult = for {
//   optionValue <- option
//   iterableValue <- iterable
//} yield optionValue + iterableValue

// It does compile when adding a .toIterable on the option
val myResult = for {
  optionValue <- option.toIterable
  iterableValue <- iterable
} yield optionValue + iterableValue
// myResult: Iterable[Int] = List(2, 3, 4, 5)
```

Una pequeña nota: si hubiéramos definido nuestra comprensión para la otra manera, la comprensión se compilaría ya que nuestra opción se convertiría implícitamente. Por esa razón, es útil agregar siempre este `.toIterable` (o la función correspondiente según la colección que esté usando) para `.toIterable` coherencia.

Lea Clase de opción en línea: https://riptutorial.com/es/scala/topic/2293/clase-de-opcion

Capítulo 7: Clases de casos

Sintaxis

- clase de caso Foo () // Las clases de caso sin parámetros deben tener una lista vacía
- clase de caso Foo (a1: A1, ..., aN: AN) // Crear una clase de caso con los campos a1 ... aN
- objeto case Bar // Crear una clase de caso singleton

Examples

Igualdad de clase de caso

Una característica proporcionada de forma gratuita por clases de casos es un método de generación automática de `equals` que verifica la igualdad de valores de todos los campos de miembros individuales en lugar de solo verificar la igualdad de referencia de los objetos.

Con las clases ordinarias:

```
class Foo(val i: Int)
val a = new Foo(3)
val b = new Foo(3)
println(a == b)// "false" because they are different objects
```

Con clases de casos:

```
case class Foo(i: Int)
val a = Foo(3)
val b = Foo(3)
println(a == b)// "true" because their members have the same value
```

Códigos generados

El modificador de `case` hace que el compilador de Scala genere automáticamente un código común para la clase. Implementar este código manualmente es tedioso y una fuente de errores. La siguiente definición de clase de caso:

```
case class Person(name: String, age: Int)
```

... se generará automáticamente el siguiente código:

```
class Person(val name: String, val age: Int)
  extends Product with Serializable
{
  def copy(name: String = this.name, age: Int = this.age): Person =
    new Person(name, age)

  def productArity: Int = 2
```

```
  def productElement(i: Int): Any = i match {
    case 0 => name
    case 1 => age
    case _ => throw new IndexOutOfBoundsException(i.toString)
  }

  def productIterator: Iterator[Any] =
    scala.runtime.ScalaRunTime.typedProductIterator(this)

  def productPrefix: String = "Person"

  def canEqual(obj: Any): Boolean = obj.isInstanceOf[Person]

  override def hashCode(): Int = scala.runtime.ScalaRunTime._hashCode(this)

  override def equals(obj: Any): Boolean = this.eq(obj) || obj match {
    case that: Person => this.name == that.name && this.age == that.age
    case _ => false
  }

  override def toString: String =
    scala.runtime.ScalaRunTime._toString(this)
}
```

El modificador de `case` también genera un objeto complementario:

```
object Person extends AbstractFunction2[String, Int, Person] with Serializable {
  def apply(name: String, age: Int): Person = new Person(name, age)

  def unapply(p: Person): Option[(String, Int)] =
    if(p == null) None else Some((p.name, p.age))
}
```

Cuando se aplica a un `object` , el modificador de `case` tiene efectos similares (aunque menos dramáticos). Aquí las principales ganancias son una implementación de `toString` y un valor de `hashCode` que es consistente en todos los procesos. Tenga en cuenta que los objetos de caso (correctamente) utilizan la igualdad de referencia:

```
object Foo extends Product with Serializable {
  def productArity: Int = 0

  def productIterator: Iterator[Any] =
    scala.runtime.ScalaRunTime.typedProductIterator(this)

  def productElement(i: Int): Any =
    throw new IndexOutOfBoundsException(i.toString)

  def productPrefix: String = "Foo"

  def canEqual(obj: Any): Boolean = obj.isInstanceOf[this.type]

  override def hashCode(): Int = 70822 // "Foo".hashCode()

  override def toString: String = "Foo"
}
```

Todavía es posible implementar manualmente métodos que de otra manera serían proporcionados por el modificador de `case` tanto en la clase en sí como en su objeto complementario.

Fundamentos de clase de caso

En comparación con las clases regulares, la notación de clases de casos brinda varios beneficios:

- Todos los argumentos del constructor son `public` y se puede acceder a ellos en objetos inicializados (normalmente este no es el caso, como se muestra aquí):

```
case class Dog1(age: Int)
val x = Dog1(18)
println(x.age) // 18 (success!)

class Dog2(age: Int)
val x = new Dog2(18)
println(x.age) // Error: "value age is not a member of Dog2"
```

- Proporciona una implementación para los siguientes métodos: `toString` , `equals` , `hashCode` (basado en propiedades), `copy` , `apply` y no `unapply` :

```
case class Dog(age: Int)
val d1 = Dog(10)
val d2 = d1.copy(age = 15)
```

- Proporciona un mecanismo conveniente para la coincidencia de patrones:

```
sealed trait Animal // `sealed` modifier allows inheritance within current build-unit
only
case class Dog(age: Int) extends Animal
case class Cat(owner: String) extends Animal
val x: Animal = Dog(18)
x match {
    case Dog(x) => println(s"It's a $x years old dog.")
    case Cat(x) => println(s"This cat belongs to $x.")
}
```

Clases de casos e inmutabilidad

El compilador de Scala prefija cada argumento en la lista de parámetros por defecto con `val` . Esto significa que, por defecto, las clases de casos son inmutables. Cada parámetro recibe un método de acceso, pero no hay métodos mutadores. Por ejemplo:

```
case class Foo(i: Int)

val fooInstance = Foo(1)
val j = fooInstance.i       // get
fooInstance.i = 2           // compile-time exception (mutation: reassignment to val)
```

Declarar un parámetro en una clase de caso como `var` anula el comportamiento predeterminado y

hace que la clase de caso sea mutable:

```
case class Bar(var i: Int)

val barInstance = Bar(1)
val j = barInstance.i       // get
barInstance.i = 2           // set
```

Otra instancia cuando una clase de caso es 'mutable' es cuando el valor en la clase de caso es mutable:

```
import scala.collection._

case class Bar(m: mutable.Map[Int, Int])

val barInstance = Bar(mutable.Map(1 -> 2))
barInstance.m.update(1, 3)                  // mutate m
barInstance                                 // Bar(Map(1 -> 3)
```

Tenga en cuenta que la 'mutación' que está ocurriendo aquí está en el mapa que `m` apunta, no `m` sí. Por lo tanto, si algún otro objeto tuviera `m` como miembro, vería el cambio también. Observe cómo en el siguiente ejemplo, cambiar la `instanceA` también cambia la `instanceB` :

```
import scala.collection.mutable

case class Bar(m: mutable.Map[Int, Int])

val m = mutable.Map(1 ->2)
val barInstanceA = Bar(m)
val barInstanceB = Bar(m)
barInstanceA.m.update(1,3)
barInstanceA  // Bar = Bar(Map(1 -> 3))
barInstanceB  // Bar = Bar(Map(1 -> 3))
m  // scala.collection.mutable.Map[Int,Int] = Map(1 -> 3)
```

Crear una copia de un objeto con ciertos cambios

Las clases de casos proporcionan un método de `copy` que crea un nuevo objeto que comparte los mismos campos que el anterior, con ciertos cambios.

Podemos usar esta función para crear un nuevo objeto a partir de uno anterior que tenga algunas de las mismas características. Esta clase de caso simple demuestra esta característica:

```
case class Person(firstName: String, lastName: String, grade: String, subject: String)
val putu = Person("Putu", "Kevin", "A1", "Math")
val mark = putu.copy(firstName = "Ketut", lastName = "Mark")
// mark: People = People(Ketut,Mark,A1,Math)
```

En este ejemplo se puede ver que los dos objetos comparten características similares (`grade = A1` , `subject = Math`), salvo que se han especificado en la copia (`firstName` y `lastName`).

Clases de caso de un solo elemento para seguridad de tipos

Para lograr la seguridad de tipos, a veces queremos evitar el uso de tipos primitivos en nuestro dominio. Por ejemplo, imagine una `Person` con un `name` . Típicamente, codificaríamos el `name` como una `String` . Sin embargo, no sería difícil de mezclar una `String` que representa una `Person` 's `name` con una `String` que representa un mensaje de error:

```
def logError(message: ErrorMessage): Unit = ???
case class Person(name: String)
val maybeName: Either[String, String] = ??? // Left is error, Right is name
maybeName.foreach(logError) // But that won't stop me from logging the name as an error!
```

Para evitar tales riesgos, puede codificar los datos de esta manera:

```
case class PersonName(value: String)
case class ErrorMessage(value: String)
case class Person(name: PersonName)
```

y ahora nuestro código no se compilará si mezclamos `PersonName` con `ErrorMessage` , o incluso una `String` normal.

```
val maybeName: Either[ErrorMessage, PersonName] = ???
maybeName.foreach(reportError) // ERROR: tried to pass PersonName; ErrorMessage expected
maybeName.swap.foreach(reportError) // OK
```

Pero esto conlleva una pequeña sobrecarga en el tiempo de ejecución, ya que ahora tenemos que encuadrar / desempaquetar `String` a / desde sus contenedores `PersonName` . Para evitar esto, uno puede hacer que las clases de valores `PersonName` y `ErrorMessage` :

```
case class PersonName(val value: String) extends AnyVal
case class ErrorMessage(val value: String) extends AnyVal
```

Lea Clases de casos en línea: https://riptutorial.com/es/scala/topic/1022/clases-de-casos

Capítulo 8: Clases de tipo

Observaciones

Para evitar problemas de serialización, particularmente en entornos distribuidos (por ejemplo, Apache Spark), es una buena práctica implementar el rasgo `Serializable` para instancias de clase de tipo.

Examples

Tipo de clase simple

Una clase de tipo es simplemente un `trait` con uno o más parámetros de tipo:

```
trait Show[A] {
  def show(a: A): String
}
```

En lugar de ampliar una clase de tipo, se proporciona una instancia implícita de la clase de tipo para cada tipo admitido. La colocación de estas implementaciones en el objeto complementario de la clase de tipo permite que la resolución implícita funcione sin ninguna importación especial:

```
object Show {
  implicit val intShow: Show[Int] = new Show {
    def show(x: Int): String = x.toString
  }

  implicit val dateShow: Show[java.util.Date] = new Show {
    def show(x: java.util.Date): String = x.getTime.toString
  }

  // ..etc
}
```

Si desea garantizar que un parámetro genérico pasado a una función tiene una instancia de una clase de tipo, use parámetros implícitos:

```
def log[A](a: A)(implicit showInstance: Show[A]): Unit = {
  println(showInstance.show(a))
}
```

También puedes usar un enlace de contexto :

```
def log[A: Show](a: A): Unit = {
  println(implicitly[Show[A]].show(a))
}
```

Llame al método de `log` anterior como cualquier otro método. No se compilará si no se puede

encontrar `Show[A]` implementación implícita de `Show[A]` para la `A` que se pasa al `log`

```
log(10) // prints: "10"
log(new java.util.Date(1469491668401L) // prints: "1469491668401"
log(List(1,2,3)) // fails to compile with
                 // could not find implicit value for evidence parameter of type
Show[List[Int]]
```

Este ejemplo implementa la clase de tipo `Show` . Esta es una clase de tipo común utilizada para convertir instancias arbitrarias de tipos arbitrarios en `String` . Aunque cada objeto tiene un método `toString` , no siempre está claro si `toString` se define o no de una manera útil. Con el uso de la clase de tipo `Show` , puede garantizar que todo lo que se pase al `log` tenga una conversión bien definida a `String` .

Extendiendo una Clase Tipo

Este ejemplo discute la extensión de la siguiente clase de tipos.

```
trait Show[A] {
  def show: String
}
```

Para hacer que una clase que *usted* controla (y está escrita en Scala) extienda la clase de tipo, agregue un implícito a su objeto complementario. Vamos a mostrar cómo podemos obtener la clase `Person` de este ejemplo para ampliar `Show` :

```
class Person(val fullName: String) {
  def this(firstName: String, lastName: String) = this(s"$firstName $lastName")
}
```

Podemos hacer que esta clase amplíe `Show` agregando un implícito al objeto complementario de `Person` :

```
object Person {
  implicit val personShow: Show[Person] = new Show {
    def show(p: Person): String = s"Person(${p.fullname})"
  }
}
```

Un objeto complementario *debe estar en el mismo archivo* que la clase, por lo que necesita la `class Person` y el `object Person` en el mismo archivo.

Para hacer una clase que no controla, o no está escrita en Scala, extienda la clase de tipo, agregue un implícito al objeto compañero de la clase de tipo, como se muestra en el ejemplo de Clase de tipo simple .

Si no controla ni la clase ni la clase de tipo, cree un implícito como el de arriba en cualquier lugar e `import` . Usando el método de `log` en el ejemplo de clase de tipo simple :

```
object MyShow {
```

```
  implicit val personShow: Show[Person] = new Show {
    def show(p: Person): String = s"Person(${p.fullname})"
  }
}

def logPeople(persons: Person*): Unit = {
  import MyShow.personShow
  persons foreach { p => log(p) }
}
```

Añadir funciones de clase de tipo a los tipos

La implementación de Scala de las clases de tipos es bastante detallada. Una forma de reducir la verbosidad es introducir las llamadas "clases de operación". Estas clases envolverán automáticamente una variable / valor cuando se importen para ampliar la funcionalidad.

Para ilustrar esto, primero creamos una clase de tipo simple:

```
// The mathematical definition of "Addable" is "Semigroup"
trait Addable[A] {
  def add(x: A, y: A): A
}
```

A continuación vamos a implementar el rasgo (instanciar la clase de tipo):

```
object Instances {

  // Instance for Int
  // Also called evidence object, meaning that this object saw that Int learned how to be
added
  implicit object addableInt extends Addable[Int] {
    def add(x: Int, y: Int): Int = x + y
  }

  // Instance for String
  implicit object addableString extends Addable[String] {
    def add(x: String, y: String): String = x + y
  }

}
```

En este momento sería muy incómodo utilizar nuestras instancias de Addable:

```
import Instances._
val three = addableInt.add(1,2)
```

Preferiríamos simplemente escribir escribir `1.add(2)` . Por lo tanto, crearemos una "Clase de operación" (también llamada "Clase de Operaciones") que siempre se ajustará a un tipo que implementa `Addable` .

```
object Ops {
  implicit class AddableOps[A](self: A)(implicit A: Addable[A]) {
    def add(other: A): A = A.add(self, other)
```

```
  }
}
```

Ahora podemos usar nuestra nueva función `add` como si fuera parte de `Int` y `String` :

```
object Main {

  import Instances._ // import evidence objects into this scope
  import Ops._       // import the wrappers

  def main(args: Array[String]): Unit = {

    println(1.add(5))
    println("mag".add("net"))
    // println(1.add(3.141)) // Fails because we didn't create an instance for Double

  }
}
```

Las clases de "operaciones" se pueden crear automáticamente mediante macros en la biblioteca de simulacros :

```
import simulacrum._

@typeclass trait Addable[A] {
  @op("|+|") def add(x: A, y: A): A
}
```

Lea Clases de tipo en línea: https://riptutorial.com/es/scala/topic/3835/clases-de-tipo

Capítulo 9: Clases y objetos

Sintaxis

- `class MyClass{} // curly braces are optional here as class body is empty`
- `class MyClassWithMethod {def method: MyClass = ???}`
- `new MyClass() //Instantiate`
- `object MyObject // Singleton object`
- `class MyClassWithGenericParameters[V1, V2](vl: V1, i: Int, v2: V2)`
- `class MyClassWithImplicitFieldCreation[V1](val v1: V1, val i: Int)`
- `new MyClassWithGenericParameters(2.3, 4, 5)` o con un tipo diferente: `new MyClassWithGenericParameters[Double, Any](2.3, 4, 5)`
- `class MyClassWithProtectedConstructor protected[my.pack.age](s: String)`

Examples

Instancia de instancias de clase

Una clase en Scala es un "plano" de una instancia de clase. Una instancia contiene el estado y el comportamiento definidos por esa clase. Para declarar una clase:

```
class MyClass{}  // curly braces are optional here as class body is empty
```

Se puede crear una instancia de una instancia usando una `new` palabra clave:

```
var instance = new MyClass()
```

o:

```
var instance = new MyClass
```

Los paréntesis son opcionales en Scala para crear objetos de una clase que tiene un constructor sin argumentos. Si un constructor de clase toma argumentos:

```
class MyClass(arg : Int)       // Class definition
var instance = new MyClass(2)  // Instance instantiation
instance.arg                   // not allowed
```

Aquí `MyClass` requiere un argumento `Int` , que solo puede usarse internamente para la clase. `arg` no se puede acceder fuera de `MyClass` menos que se declare como un campo:

```
class MyClass(arg : Int){
    val prop = arg  // Class field declaration
}

var obj = new MyClass(2)
obj.prop     // legal statement
```

Alternativamente se puede declarar público en el constructor:

```
class MyClass(val arg : Int)   // Class definition with arg declared public
var instance = new MyClass(2)  // Instance instantiation
instance.arg                   //arg is now visible to clients
```

Clase de instanciación sin parámetro: {} vs ()

Digamos que tenemos una clase MyClass sin argumento de constructor:

```
class MyClass
```

En Scala podemos crear una instancia usando la siguiente sintaxis:

```
val obj = new MyClass()
```

O simplemente podemos escribir:

```
val obj = new MyClass
```

Pero, si no se presta atención, en algunos casos, el paréntesis opcional puede producir algún comportamiento inesperado. Supongamos que queremos crear una tarea que debería ejecutarse en un hilo separado. A continuación se muestra el código de ejemplo:

```
val newThread = new Thread { new Runnable {
        override def run(): Unit = {
            // perform task
            println("Performing task.")
        }
      }
    }

newThread.start   // prints no output
```

Podemos pensar que este código de ejemplo, si se ejecuta, imprimirá la `Performing task.` , pero para nuestra sorpresa, no imprimirá nada. Veamos que está pasando aquí. Si observa detenidamente, hemos utilizado llaves `{}` , justo después del `new Thread` . Creó una clase anónima que extiende `Thread` :

```
val newThread = new Thread {
  //creating anonymous class extending Thread
}
```

Y luego, en el cuerpo de esta clase anónima, definimos nuestra tarea (nuevamente creando una clase `Runnable` implementa la interfaz `Runnable`). Así que podríamos haber pensado que usamos el constructor `public Thread(Runnable target)` pero de hecho (al ignorar opcional `()`) usamos el constructor `public Thread()` sin nada definido en el cuerpo del método `run()` . Para corregir el problema, necesitamos usar paréntesis en lugar de llaves.

```
val newThread = new Thread ( new Runnable {
        override def run(): Unit = {
            // perform task
            println("Performing task.")
        }
      }
    )
```

En otras palabras, aquí `{}` y `()` no son *intercambiables* .

Objetos singleton y acompañantes

Objetos Singleton

Scala admite miembros estáticos, pero no de la misma manera que Java. Scala proporciona una alternativa a esto llamada *Objetos Singleton* . Los objetos Singleton son similares a una clase normal, excepto que no se pueden crear instancias con la `new` palabra clave. A continuación se muestra una muestra de clase singleton:

```
object Factorial {
    private val cache = Map[Int, Int]()
    def getCache = cache
}
```

Tenga en cuenta que hemos utilizado la palabra clave `object` para definir el objeto singleton (en lugar de 'class' o 'trait'). Como los objetos singleton no pueden ser instanciados, no pueden tener parámetros. El acceso a un objeto singleton se ve así:

```
Factorial.getCache() //returns the cache
```

Tenga en cuenta que esto se ve exactamente como acceder a un método estático en una clase de Java.

Objetos Acompañantes

En Scala, los objetos singleton pueden compartir el nombre de una clase correspondiente. En tal escenario, el objeto singleton se conoce como un *objeto complementario* . Por ejemplo, debajo de la clase `Factorial` se define, y un objeto complementario (también llamado `Factorial`) se define debajo de ella. Por convención, los objetos complementarios se definen en el mismo archivo que su clase complementaria.

```
class Factorial(num : Int) {

  def fact(num : Int) : Int = if (num <= 1) 1 else (num * fact(num - 1))

  def calculate() : Int = {
    if (!Factorial.cache.contains(num)) {    // num does not exists in cache
      val output = fact(num) // calculate factorial
      Factorial.cache += (num -> output)     // add new value in cache
    }
```

```
    Factorial.cache(num)
  }
}

object Factorial {
  private val cache = scala.collection.mutable.Map[Int, Int]()
}

val factfive = new Factorial(5)
factfive.calculate  // Calculates the factorial of 5 and stores it
factfive.calculate  // uses cache this time
val factfiveagain = new Factorial(5)
factfiveagain.calculate  // Also uses cache
```

En este ejemplo, estamos usando un `cache` privado para almacenar factorial de un número para ahorrar tiempo de cálculo para números repetidos.

Aquí el `object Factorial` es un objeto compañero y la `class Factorial` es su clase compañera correspondiente. Los objetos y clases complementarios pueden acceder a `private` miembros `private` cada uno. En el ejemplo anterior, la clase `Factorial` está accediendo al miembro de `cache` privado de su objeto complementario.

Tenga en cuenta que una nueva instanciación de la clase seguirá utilizando el mismo objeto complementario, por lo que cualquier modificación de las variables miembro de ese objeto se transferirá.

Objetos

Mientras que las clases son más como planos, los objetos son estáticos (es decir, ya están instanciados):

```
object Dog {
    def bark: String = "Raf"
}

Dog.bark() // yields "Raf"
```

A menudo se usan como un compañero de una clase, le permiten escribir:

```
class Dog(val name: String) {

}

object Dog {
    def apply(name: String): Dog = new Dog(name)
}

val dog = Dog("Barky") // Object
val dog = new Dog("Barky") // Class
```

Comprobación del tipo de instancia

Verificación de tipo : `variable.isInstanceOf[Type]`

Con coincidencia de patrones (no tan útil en esta forma):

```
variable match {
  case _: Type => true
  case _ => false
}
```

Tanto `isInstanceOf` como la coincidencia de patrones están verificando solo el tipo de objeto, no su parámetro genérico (sin reificación de tipo), excepto los arreglos:

```
val list: List[Any] = List(1, 2, 3)             //> list  : List[Any] = List(1, 2, 3)

val upcasting = list.isInstanceOf[Seq[Int]]     //> upcasting  : Boolean = true

val shouldBeFalse = list.isInstanceOf[List[String]]
                                                //> shouldBeFalse  : Boolean = true
```

Pero

```
val chSeqArray: Array[CharSequence] = Array("a") //> chSeqArray  : Array[CharSequence] =
Array(a)
val correctlyReified = chSeqArray.isInstanceOf[Array[String]]
                                                //> correctlyReified  : Boolean = false


val stringIsACharSequence: CharSequence = ""    //> stringIsACharSequence  : CharSequence = ""

val sArray = Array("a")                         //> sArray  : Array[String] = Array(a)
val correctlyReified = sArray.isInstanceOf[Array[String]]
                                                //> correctlyReified  : Boolean = true

//val arraysAreInvariantInScala: Array[CharSequence] = sArray
//Error: type mismatch;  found   : Array[String]  required: Array[CharSequence]
//Note: String <: CharSequence, but class Array is invariant in type T.
//You may wish to investigate a wildcard type such as `_ <: CharSequence`. (SLS 3.2.10)
//Workaround:
val arraysAreInvariantInScala: Array[_ <: CharSequence] = sArray
                                                //> arraysAreInvariantInScala  : Array[_ <:
CharSequence] = Array(a)


val arraysAreCovariantOnJVM = sArray.isInstanceOf[Array[CharSequence]]
                                                //> arraysAreCovariantOnJVM  : Boolean = true
```

Tipo casting : `variable.asInstanceOf[Type]`

Con la coincidencia de patrones :

```
variable match {
  case _: Type => true
}
```

Ejemplos:

```
  val x = 3                                       //> x  : Int = 3
  x match {
    case _: Int => true//better: do something
    case _ => false
  }                                               //> res0: Boolean = true

  x match {
    case _: java.lang.Integer => true//better: do something
    case _ => false
  }                                               //> res1: Boolean = true

  x.isInstanceOf[Int]                             //> res2: Boolean = true

  //x.isInstanceOf[java.lang.Integer]//fruitless type test: a value of type Int cannot also be
a Integer

  trait Valuable { def value: Int}
  case class V(val value: Int) extends Valuable

  val y: Valuable = V(3)                          //> y  : Valuable = V(3)
  y.isInstanceOf[V]                               //> res3: Boolean = true
  y.asInstanceOf[V]                               //> res4: V = V(3)
```

Observación: esto es solo sobre el comportamiento en la JVM, en otras plataformas (JS, nativo) el tipo de conversión / comprobación podría comportarse de manera diferente.

Constructores

Constructor primario

En Scala el constructor primario es el cuerpo de la clase. El nombre de la clase va seguido de una lista de parámetros, que son los argumentos del constructor. (Como con cualquier función, se puede omitir una lista de parámetros vacía).

```
class Foo(x: Int, y: String) {
    val xy: String = y * x
    /* now xy is a public member of the class */
}

class Bar {
    ...
}
```

Los parámetros de construcción de una instancia no son accesibles fuera de su cuerpo de constructor a menos que estén marcados como miembros de instancia por la palabra clave `val` :

```
class Baz(val z: String)
// Baz has no other members or methods, so the body may be omitted

val foo = new Foo(4, "ab")
val baz = new Baz("I am a baz")
foo.x // will not compile: x is not a member of Foo
foo.xy // returns "abababab": xy is a member of Foo
baz.z // returns "I am a baz": z is a member of Baz
val bar0 = new Bar
```

```
val bar1 = new Bar() // Constructor parentheses are optional here
```

Cualquier operación que deba realizarse cuando se crea una instancia de un objeto se escribe directamente en el cuerpo de la clase:

```
class DatabaseConnection
    (host: String, port: Int, username: String, password: String) {
    /* first connect to the DB, or throw an exception */
    private val driver = new AwesomeDB.Driver()
    driver.connect(host, port, username, password)
    def isConnected: Boolean = driver.isConnected
    ...
}
```

Tenga en cuenta que se considera una buena práctica poner la menor cantidad posible de efectos secundarios en el constructor; en lugar del código anterior, se debe considerar tener métodos de `connect` y `disconnect` para que el código del consumidor sea responsable de programar la IO.

Constructores Auxiliares

Una clase puede tener constructores adicionales llamados 'constructores auxiliares'. Estas están definidas por definiciones de constructor en la forma `def this(...) = e` , donde `e` debe invocar a otro constructor:

```
class Person(val fullName: String) {
  def this(firstName: String, lastName: String) = this(s"$firstName $lastName")
}

// usage:
new Person("Grace Hopper").fullName // returns Grace Hopper
new Person("Grace", "Hopper").fullName // returns Grace Hopper
```

Esto implica que cada constructor puede tener un modificador diferente: solo algunos pueden estar disponibles públicamente:

```
class Person private(val fullName: String) {
  def this(firstName: String, lastName: String) = this(s"$firstName $lastName")
}

new Person("Ada Lovelace") // won't compile
new Person("Ada", "Lovelace") // compiles
```

De esta manera puede controlar cómo el código de consumidor puede instanciar la clase.

Lea Clases y objetos en línea: https://riptutorial.com/es/scala/topic/2047/clases-y-objetos

Capítulo 10: Colecciones

Examples

Ordenar una lista

Suponiendo que en la siguiente lista podemos clasificar una variedad de formas.

```
val names = List("Kathryn", "Allie", "Beth", "Serin", "Alana")
```

El comportamiento predeterminado de `sorted()` es usar `math.Ordering` , que para cadenas da como resultado una ordenación lexográfica :

```
names.sorted
// results in: List(Alana, Allie, Beth, Kathryn, Serin)
```

`sortWith` permite proporcionar su propio pedido utilizando una función de comparación:

```
names.sortWith(_.length < _.length)
// results in: List(Beth, Allie, Serin, Alana, Kathryn)
```

`sortBy` permite proporcionar una función de transformación:

```
//A set of vowels to use
val vowels = Set('a', 'e', 'i', 'o', 'u')

//A function that counts the vowels in a name
def countVowels(name: String) = name.count(l => vowels.contains(l.toLower))

//Sorts by the number of vowels
names.sortBy(countVowels)
//result is: List(Kathryn, Beth, Serin, Allie, Alana)
```

Siempre puede revertir una lista, o una lista ordenada, usando `reverse:

```
names.sorted.reverse
//results in: List(Serin, Kathryn, Beth, Allie, Alana)
```

Las listas también se pueden ordenar utilizando el método Java `java.util.Arrays.sort` y su envoltorio Scala `scala.util.Sorting.quickSort`

```
java.util.Arrays.sort(data)
scala.util.Sorting.quickSort(data)
```

Estos métodos pueden mejorar el rendimiento al ordenar colecciones más grandes si se pueden evitar las conversiones de colección y el desempaquetado / boxeo. Para una discusión más detallada sobre las diferencias de rendimiento, lea acerca de Scala Collection ordenados,

sortWith y sortBy Performance .

Crear una lista que contenga n copias de x

Para crear una colección de `n` copias de algún objeto `x` , use el método de relleno . Este ejemplo crea una `List` , pero esto puede funcionar con otras colecciones para las que el `fill` tiene sentido:

```
// List.fill(n)(x)
scala > List.fill(3)("Hello World")
res0: List[String] = List(Hello World, Hello World, Hello World)
```

Lista y Vector Cheatsheet

> Ahora es una práctica recomendada utilizar `Vector` lugar de `List` porque las implementaciones tienen un mejor rendimiento Las características de rendimiento se pueden encontrar aquí . `Vector` se puede utilizar donde se utiliza la `List` .

Creación de listas

```
List[Int]()          // Declares an empty list of type Int
List.empty[Int]      // Uses `empty` method to declare empty list of type Int
Nil                  // A list of type Nothing that explicitly has nothing in it

List(1, 2, 3)        // Declare a list with some elements
1 :: 2 :: 3 :: Nil   // Chaining element prepending to an empty list, in a LISP-style
```

Tomar elemento

```
List(1, 2, 3).headOption // Some(1)
List(1, 2, 3).head       // 1

List(1, 2, 3).lastOption // Some(3)
List(1, 2, 3).last       // 3, complexity is O(n)

List(1, 2, 3)(1)         // 2, complexity is O(n)
List(1, 2, 3)(3)         // java.lang.IndexOutOfBoundsException: 4
```

Elementos Prependidos

```
0 :: List(1, 2, 3)       // List(0, 1, 2, 3)
```

Anexar elementos

```
List(1, 2, 3) :+ 4       // List(1, 2, 3, 4), complexity is O(n)
```

Unirse (concatenar) listas

```
List(1, 2) ::: List(3, 4) // List(1, 2, 3, 4)
List.concat(List(1,2), List(3, 4)) // List(1, 2, 3, 4)
List(1, 2) ++ List(3, 4)  // List(1, 2, 3, 4)
```

Operaciones comunes

```
List(1, 2, 3).find(_ == 3)                     // Some(3)
List(1, 2, 3).map(_ * 2)                       // List(2, 4, 6)
List(1, 2, 3).filter(_ % 2 == 1)               // List(1, 3)
List(1, 2, 3).fold(0)((acc, i) => acc + i * i) // 1 * 1 + 2 * 2 + 3 * 3 = 14
List(1, 2, 3).foldLeft("Foo")(_ + _.toString)  // "Foo123"
List(1, 2, 3).foldRight("Foo")(_ + _.toString) // "123Foo"
```

Mapa de la colección Cheatsheet

> Tenga en cuenta que esto se refiere a la creación de una colección de tipo `Map` , que es distinta del método de `map` .

Creación de mapas

```
Map[String, Int]()
val m1: Map[String, Int] = Map()
val m2: String Map Int = Map()
```

Un mapa puede considerarse una colección de `tuples` para la mayoría de las operaciones, donde el primer elemento es la clave y el segundo es el valor.

```
val l = List(("a", 1), ("b", 2), ("c", 3))
val m = l.toMap                               // Map(a -> 1, b -> 2, c -> 3)
```

Obtener elemento

```
val m = Map("a" -> 1, "b" -> 2, "c" -> 3)

m.get("a")  // Some(1)
m.get("d")  // None
m("a")      // 1
m("d")      // java.util.NoSuchElementException: key not found: d

m.keys      // Set(a, b, c)
m.values    // MapLike(1, 2, 3)
```

Añadir elemento (s)

```
Map("a" -> 1, "b" -> 2) + ("c" -> 3)              // Map(a -> 1, b -> 2, c -> 3)
Map("a" -> 1, "b" -> 2) + ("a" -> 3)              // Map(a -> 3, b -> 2)
Map("a" -> 1, "b" -> 2) ++ Map("b" -> 3, "c" -> 4) // Map(a -> 1, b -> 3, c -> 4)
```

Operaciones comunes

En las operaciones donde ocurre una iteración sobre un mapa (`map` , `find` , `forEach` , etc.), los elementos de la colección son `tuples` . El parámetro de función puede usar los accesores de tupla (`_1` , `_2`), o una función parcial con un bloque de caso:

```
m.find(_._1 == "a")  // Some((a,1))
```

```
m.map {
  case (key, value) => (value, key)
}                    // Map(1 -> a, 2 -> b, 3 -> c)
m.filter(_._2 == 2)  // Map(b -> 2)
m.foldLeft(0){
  case (acc, (key, value: Int)) => acc + value
}                    // 6
```

Mapa y filtro sobre una colección

Mapa

El 'Mapeo' a través de una colección usa la función de `map` para transformar cada elemento de esa colección de una manera similar. La sintaxis general es:

```
val someFunction: (A) => (B) = ???
collection.map(someFunction)
```

Puede proporcionar una función anónima:

```
collection.map((x: T) => /*Do something with x*/)
```

Multiplicando números enteros por dos

```
// Initialize
val list = List(1,2,3)
// list: List[Int] = List(1, 2, 3)

// Apply map
list.map((item: Int) => item*2)
// res0: List[Int] = List(2, 4, 6)

// Or in a more concise way
list.map(_*2)
// res1: List[Int] = List(2, 4, 6)
```

Filtrar

`filter` se utiliza cuando desea excluir o 'filtrar' ciertos elementos de una colección. Al igual que con `map` , la sintaxis general toma una función, pero esa función debe devolver un `Boolean` :

```
val someFunction: (a) => Boolean = ???
collection.filter(someFunction)
```

Puede proporcionar una función anónima directamente:

```
collection.filter((x: T) => /*Do something that returns a Boolean*/)
```

Comprobando los números de pares

```
val list = 1 to 10 toList
// list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

// Filter out all elements that aren't evenly divisible by 2
list.filter((item: Int) => item % 2==0)
// res0: List[Int] = List(2, 4, 6, 8, 10)
```

Más ejemplos de mapas y filtros

```
case class Person(firstName: String,
                  lastName: String,
                  title: String)

// Make a sequence of people
val people = Seq(
  Person("Millie", "Fletcher", "Mrs"),
  Person("Jim", "White", "Mr"),
  Person("Jenny", "Ball", "Miss") )


// Make labels using map
val labels = people.map( person =>
  s"${person.title}. ${person.lastName}"
)

// Filter the elements beginning with J
val beginningWithJ = people.filter(_.firstName.startsWith("J"))

// Extract first names and concatenate to a string
val firstNames = people.map(_.firstName).reduce( (a, b) => a + "," + b )
```

Introducción a las colecciones de Scala

El marco de Scala Collections, según sus autores , está diseñado para ser fácil de usar, conciso, seguro, rápido y universal.

El marco se compone de rasgos de Scala que están diseñados para ser bloques de construcción para crear colecciones. Para obtener más información sobre estos bloques de construcción, lea el resumen oficial de las colecciones de Scala .

Estas colecciones integradas se separan en los paquetes inmutables y mutables. Por defecto, se utilizan las versiones inmutables. Construir una `List()` (sin importar nada) construirá una lista *inmutable* .

Una de las características más poderosas del marco es la interfaz consistente y fácil de usar a través de colecciones afines. Por ejemplo, sumar todos los elementos en una colección es igual para Listas, Conjuntos, Vectores, Seqs y Arrays:

```
val numList = List[Int](1, 2, 3, 4, 5)
```

```
numList.reduce((n1, n2) => n1 + n2)  // 15

val numSet = Set[Int](1, 2, 3, 4, 5)
numSet.reduce((n1, n2) => n1 + n2)   // 15

val numArray = Array[Int](1, 2, 3, 4, 5)
numArray.reduce((n1, n2) => n1 + n2) // 15
```

Estos tipos afines heredan del rasgo `Traversable` .

> Ahora es una práctica recomendada utilizar `Vector` lugar de `List` porque las implementaciones tienen un mejor rendimiento Las características de rendimiento se pueden encontrar aquí . `Vector` se puede utilizar donde se utiliza la `List` .

Tipos transitables

Clases de colección que tienen el `Traversable` rasgo implementar `foreach` y heredar muchos métodos para realizar operaciones comunes a las colecciones, que todos funcionan de forma idéntica. Las operaciones más comunes se enumeran aquí:

- Mapa - `map` , `flatMap` y `collect` producir nuevas colecciones mediante la aplicación de una función a cada elemento de la colección original.

```
List(1, 2, 3).map(num => num * 2) // double every number = List(2, 4, 6)

// split list of letters into individual strings and put them into the same list
List("a b c", "d e").flatMap(letters => letters.split(" ")) // = List("a", "b", "c", "d", "e")
```

- Conversiones : `toList` , `toArray` y muchas otras operaciones de conversión cambian la colección actual en un tipo más específico de colección. Por lo general, estos son métodos prependidos con 'to' y el tipo más específico (es decir, 'toList' se convierte en una `List`).

```
val array: Array[Int] = List[Int](1, 2, 3).toArray // convert list of ints to array of ints
```

- Información de tamaño : `isEmpty` , `nonEmpty` , `size` y `hasDefiniteSize` son todos metadatos sobre el conjunto. Esto permite operaciones condicionales en la colección, o para que el código determine el tamaño de la colección, incluso si es infinito o discreto.

```
List().isEmpty // true
List(1).nonEmpty // true
```

- Recuperación de elementos : `head` , `last` , `find` y sus variantes de `Option` se utilizan para recuperar el primer o último elemento, o encontrar un elemento específico en la colección.

```
val list = List(1, 2, 3)
list.head // = 1
list.last // = 3
```

- Operaciones de recuperación de subcolecciones: las operaciones de `filter` , `tail` , `slice` , `drop`

y otras permiten elegir partes de la colección para seguir operando.

```
List(-2, -1, 0, 1, 2).filter(num => num > 0) // = List(1, 2)
```

- Operaciones de subdivisión : `partition` , `splitAt` , `span` y `groupBy` dividen la colección actual en partes diferentes.

```
// split numbers into < 0 and >= 0
List(-2, -1, 0, 1, 2).partition(num => num < 0) // = (List(-2, -1), List(0, 1, 2))
```

- Pruebas de elementos - `exists` , `forall` , y `count` se utilizan operaciones de control de esta colección para ver si satisface un predicado.

```
List(1, 2, 3, 4).forall(num => num > 0) // = true, all numbers are positive
List(-3, -2, -1, 1).forall(num => num < 0) // = false, not all numbers are negative
```

- Folds : `foldLeft` (`/:` `foldRight` , `foldRight` (`:\`), `reduceLeft` y `reduceRight` se utilizan para aplicar funciones binarias a elementos sucesivos de la colección. Vaya aquí para ver ejemplos de pliegues e ir aquí para ejemplos de reducción

Doblez

El método de `fold` recorre una colección, utilizando un valor inicial del acumulador y aplicando una función que utiliza cada elemento para actualizar el acumulador con éxito:

```
val nums = List(1,2,3,4,5)
var initialValue:Int = 0;
var sum = nums.fold(initialValue){
  (accumulator,currentElementBeingIterated) => accumulator + currentElementBeingIterated
}
println(sum) //prints 15 because 0+1+2+3+4+5 = 15
```

En el ejemplo anterior, se proporcionó una función anónima para `fold()` . También puede usar una función nombrada que toma dos argumentos. Teniendo esto en mi, el ejemplo anterior se puede reescribir así:

```
def sum(x: Int, y: Int) = x+ y
val nums = List(1, 2, 3, 4, 5)
var initialValue: Int = 0
val sum = nums.fold(initialValue)(sum)
println(sum) // prints 15 because 0 + 1 + 2 + 3 + 4 + 5 = 15
```

Cambiar el valor inicial afectará el resultado:

```
initialValue = 2;
sum = nums.fold(initialValue){
 (accumulator,currentElementBeingIterated) => accumulator + currentElementBeingIterated
}
println(sum) //prints 17 because 2+1+2+3+4+5 = 17
```

El método de `fold` tiene dos variantes: `foldLeft` y `foldRight` .

`foldLeft()` itera de izquierda a derecha (desde el primer elemento de la colección hasta el último en ese orden). `foldRight()` itera de derecha a izquierda (desde el último elemento hasta el primer elemento). `fold()` itera de izquierda a derecha como `foldLeft()` . De hecho, `fold()` realidad llama a `foldLeft()` internamente.

```
def fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1 = foldLeft(z)(op)
```

`fold()` , `foldLeft()` y `foldRight()` devolverá un valor que tiene el mismo tipo con el valor inicial que toma. Sin embargo, a diferencia de `foldLeft()` y `foldRight()` , el valor inicial dado para `fold()` solo puede ser del mismo tipo o un supertipo del tipo de la colección.

En este ejemplo, el orden no es relevante, por lo que puede cambiar `fold()` por `foldLeft()` o `foldRight()` y el resultado seguirá siendo el mismo. El uso de una función que sea sensible al orden alterará los resultados.

En caso de duda, prefiera `foldLeft()` sobre `foldRight()` . `foldRight()` tiene menos rendimiento.

Para cada

`foreach` es inusual entre los iteradores de colecciones porque no devuelve un resultado. En su lugar, aplica una función a cada elemento que tiene solo efectos secundarios. Por ejemplo:

```
scala> val x = List(1,2,3)
x: List[Int] = List(1, 2, 3)

scala> x.foreach { println }
1
2
3
```

La función suministrada a `foreach` puede tener cualquier tipo de retorno, pero el resultado se descartará . Normalmente se usa `foreach` cuando los efectos secundarios son deseables. Si desea transformar los datos, considere usar un `map` , un `filter` , una `for comprehension` u otra opción.

Ejemplo de descarte de resultados.

```
def myFunc(a: Int) : Int = a * 2
List(1,2,3).foreach(myFunc) // Returns nothing
```

Reducir

Los métodos `reduce()` , `reduceLeft()` y `reduceRight` son similares a los pliegues. La función pasada para reducir toma dos valores y produce un tercero. Cuando se opera en una lista, los dos primeros valores son los dos primeros valores de la lista. El resultado de la función y el siguiente valor en la lista se vuelven a aplicar a la función, produciendo un nuevo resultado. Este nuevo resultado se aplica con el siguiente valor de la lista y así sucesivamente hasta que no haya más

elementos. Se devuelve el resultado final.

```
val nums = List(1,2,3,4,5)
sum = nums.reduce({ (a, b) => a + b })
println(sum) //prints 15

val names = List("John","Koby", "Josh", "Matilda", "Zac", "Mary Poppins")

def findLongest(nameA:String, nameB:String):String = {
  if (nameA.length > nameB.length) nameA else nameB
}

def findLastAlphabetically(nameA:String, nameB:String):String = {
  if (nameA > nameB) nameA else nameB
}

val longestName:String = names.reduce(findLongest(_,_))
println(longestName) //prints Mary Poppins

//You can also omit the arguments if you want
val lastAlphabetically:String = names.reduce(findLastAlphabetically)
println(lastAlphabetically) //prints Zac
```

Existen algunas diferencias en cómo funcionan las funciones de reducción en comparación con las funciones de plegado. Son:

1. Las funciones de reducción no tienen valor acumulador inicial.
2. Las funciones de reducción no se pueden llamar en listas vacías.
3. Las funciones de reducción solo pueden devolver el tipo o supertipo de la lista.

Lea Colecciones en línea: https://riptutorial.com/es/scala/topic/686/colecciones

Capítulo 11: Colecciones paralelas

Observaciones

Las colecciones paralelas facilitan la programación paralela al ocultar los detalles de paralelización de bajo nivel. Esto hace que tomar ventaja de las arquitecturas de múltiples núcleos sea fácil. Los ejemplos de colecciones paralelas incluyen `ParArray` , `ParVector` , `mutable.ParHashMap` , `immutable.ParHashMap` y `ParRange` . Una lista completa se puede encontrar en la documentación .

Examples

Creación y uso de colecciones paralelas

Para crear una colección paralela a partir de una colección secuencial, llame al método `par` . Para crear una colección secuencial a partir de una colección paralela, llame al método `seq` . Este ejemplo muestra cómo convertir un `Vector` regular en un `ParVector` y luego de nuevo:

```
scala> val vect = (1 to 5).toVector
vect: Vector[Int] = Vector(1, 2, 3, 4, 5)

scala> val parVect = vect.par
parVect: scala.collection.parallel.immutable.ParVector[Int] = ParVector(1, 2, 3, 4, 5)

scala> parVect.seq
res0: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3, 4, 5)
```

El método `par` se puede encadenar, lo que le permite convertir una colección secuencial en una colección paralela y realizar una acción de inmediato:

```
scala> vect.map(_ * 2)
res1: scala.collection.immutable.Vector[Int] = Vector(2, 4, 6, 8, 10)

scala> vect.par.map(_ * 2)
res2: scala.collection.parallel.immutable.ParVector[Int] = ParVector(2, 4, 6, 8, 10)
```

En estos ejemplos, el trabajo en realidad se divide en varias unidades de procesamiento y luego se vuelve a unir después de que el trabajo se completa, sin la intervención del desarrollador.

Escollos

No utilice colecciones paralelas cuando los elementos de la colección deben recibirse en un orden específico.

Las colecciones paralelas realizan operaciones concurrentemente. Eso significa que todo el trabajo se divide en partes y se distribuye a diferentes procesadores. Cada procesador desconoce el trabajo realizado por otros. Si el *orden de la colección* importa, entonces el trabajo procesado

en paralelo no es determinista. (Ejecutar el mismo código dos veces puede producir resultados diferentes).

Operaciones no asociativas

Si una operación no es asociativa (si el orden de ejecución es importante), el resultado en una recopilación en paralelo no será determinista.

```
scala> val list = (1 to 1000).toList
list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10...

scala> list.reduce(_ - _)
res0: Int = -500498

scala> list.reduce(_ - _)
res1: Int = -500498

scala> list.reduce(_ - _)
res2: Int = -500498

scala> val listPar = list.par
listPar: scala.collection.parallel.immutable.ParSeq[Int] = ParVector(1, 2, 3, 4, 5, 6, 7, 8,
9, 10...

scala> listPar.reduce(_ - _)
res3: Int = -408314

scala> listPar.reduce(_ - _)
res4: Int = -422884

scala> listPar.reduce(_ - _)
res5: Int = -301748
```

Efectos secundarios

Las operaciones que tienen efectos secundarios, como `foreach` , pueden no ejecutarse como se desea en colecciones en paralelo debido a las condiciones de la carrera. Evite esto utilizando funciones que no tengan efectos secundarios, como `reduce` o `map` .

```
scala> val wittyOneLiner = Array("Artificial", "Intelligence", "is", "no", "match", "for",
"natural", "stupidity")

scala> wittyOneLiner.foreach(word => print(word + " "))
Artificial Intelligence is no match for natural stupidity

scala> wittyOneLiner.par.foreach(word => print(word + " "))
match natural is for Artificial no stupidity Intelligence

scala> print(wittyOneLiner.par.reduce{_ + " " + _})
Artificial Intelligence is no match for natural stupidity

scala> val list = (1 to 100).toList
list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15...
```

Lea Colecciones paralelas en línea: https://riptutorial.com/es/scala/topic/3882/colecciones-paralelas

Capítulo 12: Combinadores de analizador

Observaciones

Casos de ParseResult

Un `ParseResult` viene en tres sabores:

- Éxito, con un marcador en cuanto al inicio del partido y el siguiente carácter que se va a combinar.
- Fracaso, con un marcador en el inicio de donde se intentó la coincidencia. En este caso, el analizador retrocede a esa posición, donde estará cuando el análisis continúe.
- Error, que detiene el análisis. No se realiza ningún retroceso o análisis adicional.

Examples

Ejemplo básico

```
import scala.util.parsing.combinator._

class SimpleParser extends RegexParsers {
  // Define a grammar rule, turn it into a regex, and apply it the input.
  def word: Parser[String] = """[A-Z][a-z]+""".r ^^ { _.toString }
}

object SimpleParser extends SimpleParser {
  val parseAlice = parse(word, "Alice went to Alamo Square.")
  val parseBarb = parse(word, "barb went Upside Down.")
}

//Successfully finds a match
println(SimpleParser.parseAlice)
//Fails to find a match
println(SimpleParser.parseBarb)
```

La salida será la siguiente:

```
[1.6] parsed: Alice
res0: Unit = ()

[1.1] failure: string matching regex `[A-Z][a-z]+' expected but `b' found

barb went Upside Down.
^
```

`[1.6]` en el ejemplo de `Alice` indica que el inicio de la partida está en la posición `1` , y el primer carácter que queda para coincidir comienza en la posición `6` .

Lea Combinadores de analizador en línea:
https://riptutorial.com/es/scala/topic/3730/combinadores-de-analizador

Capítulo 13: Configurando Scala

Examples

En Linux a través de dpkg

En las distribuciones basadas en Debian, incluyendo Ubuntu, la forma más sencilla es usar el archivo de instalación `.deb` . Ir al sitio web de Scala . Elija la versión que desea instalar, luego desplácese hacia abajo y busque `scala-xxxdeb` .

Puedes instalar el scala deb desde la línea de comando:

```
sudo dpkg -i scala-x.x.x.deb
```

Para verificar que está instalado correctamente, en el indicador de comandos de la terminal:

```
which scala
```

La respuesta devuelta debe ser el equivalente a lo que colocó en su variable PATH. Para verificar que scala está funcionando:

```
scala
```

Esto debería iniciar el REPL de Scala e informar la versión (que, a su vez, debe coincidir con la versión que descargó).

Instalación de Ubuntu a través de descarga manual y configuración

Descargue su versión preferida de Lightbend con `curl` :

```
curl -O http://downloads.lightbend.com/scala/2.xx.x/scala-2.xx.x.tgz
```

Descomprima el archivo `tar` en `/usr/local/share` o `/opt/bin` :

```
unzip scala-2.xx.x.tgz
mv scala-2.xx.x /usr/local/share/scala
```

Agregue el `PATH` a `~/.profile` o `~/.bash_profile` o `~/.bashrc` incluyendo este texto en uno de esos archivos:

```
$SCALA_HOME=/usr/local/share/scala
export PATH=$SCALA_HOME/bin:$PATH
```

Para verificar que está instalado correctamente, en el indicador de comandos de la terminal:

```
which scala
```

La respuesta devuelta debe ser el equivalente a lo que colocó en su variable `PATH` . Para verificar que `scala` está funcionando:

```
scala
```

Esto debería iniciar el REPL de Scala e informar la versión (que, a su vez, debe coincidir con la versión que descargó).

Mac OSX a través de Macports

En computadoras Mac OSX con MacPorts instalado, abra una ventana de terminal y escriba:

```
port list | grep scala
```

Esto mostrará una lista de todos los paquetes relacionados con Scala disponibles. Para instalar uno (en este ejemplo, la versión 2.11 de Scala):

```
sudo port install scala2.11
```

(El `2.11` puede cambiar si desea instalar una versión diferente.)

Todas las dependencias se instalarán automáticamente y se actualizará su parámetro `$PATH` . Para verificar todo funcionó:

```
which scala
```

Esto le mostrará la ruta a la instalación de Scala.

```
scala
```

Esto abrirá el REPL de Scala e informará el número de versión instalada.

Lea Configurando Scala en línea: https://riptutorial.com/es/scala/topic/2921/configurando-scala

Capítulo 14: Corrientes

Observaciones

Los flujos se evalúan perezosamente, lo que significa que se pueden usar para implementar generadores, que proporcionarán o 'generarán' un nuevo elemento del tipo especificado a pedido, en lugar de antes del hecho. Esto asegura que solo se realicen los cálculos necesarios.

Examples

Uso de un flujo para generar una secuencia aleatoria

`genRandom` crea una secuencia de números aleatorios que tiene una probabilidad de uno en cuatro de terminar cada vez que se llama.

```
def genRandom: Stream[String] = {
  val random = scala.util.Random.nextFloat()
  println(s"Random value is: $random")
  if (random < 0.25) {
    Stream.empty[String]
  } else {
    ("%.3f : A random number" format random) #:: genRandom
  }
}

lazy val randos = genRandom  // getRandom is lazily evaluated as randos is iterated through

for {
  x <- randos
} println(x) // The number of times this prints is effectively randomized.
```

Tenga en cuenta la construcción `#::` , que *recurre perezosamente* : debido a que está anteponiendo el número aleatorio actual a una secuencia, no evalúa el resto de la secuencia hasta que se itera.

Corrientes infinitas a través de la recursión

Se pueden construir flujos que se refieran a sí mismos y, por lo tanto, se vuelvan infinitamente recursivos.

```
// factorial
val fact: Stream[BigInt] = 1 #:: fact.zipWithIndex.map{case (p,x)=>p*(x+1)}
fact.take(10)  // (1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880)
fact(24)       // 620448401733239439360000

// the Fibonacci series
val fib: Stream[BigInt] = 0 #:: fib.scan(1:BigInt)(_+_)
fib.take(10)  // (0, 1, 1, 2, 3, 5, 8, 13, 21, 34)
fib(124)      // 36726740705505779255899443
```

```
// random Ints between 10 and 99 (inclusive)
def rndInt: Stream[Int] = (util.Random.nextInt(90)+10) #:: rndInt
rndInt.take(10)  // (20, 95, 14, 44, 42, 78, 85, 24, 99, 85)
```

En este contexto, la diferencia entre Var, Val y Def es interesante. Como `def` cada elemento se recalcula cada vez que se hace referencia. Como valor `val` cada elemento se conserva y se reutiliza una vez calculado. Esto se puede demostrar creando un efecto secundario con cada cálculo.

```
// def with extra output per calculation
def fact: Stream[Int] = 1 #:: fact.zipWithIndex.map{case (p,x)=>print("!");p*(x+1)}
fact(5)  // !!!!!!!!!!!!!! 120
fact(4)  // !!!!!!!!!! 24
fact(7)  // !!!!!!!!!!!!!!!!!!!!!!!!!! 5040

// now as val
val fact: Stream[Int] = 1 #:: fact.zipWithIndex.map{case (p,x)=>print("!");p*(x+1)}
fact(5)  // !!!!! 120
fact(4)  // 24
fact(7)  // !! 5040
```

Esto también explica por qué la `Stream` números aleatorios no funciona como un `val` .

```
val rndInt: Stream[Int] = (util.Random.nextInt(90)+10) #:: rndInt
rndInt.take(5)  // (79, 79, 79, 79, 79)
```

Secuencia infinita auto-referente

```
// Generate stream that references itself in its evaluation
lazy val primes: Stream[Int] =
  2 #:: Stream.from(3, 2)
    .filter { i => primes.takeWhile(p => p * p <= i).forall(i % _ != 0) }
    .takeWhile(_ > 0) // prevent overflowing

// Get list of 10 primes
assert(primes.take(10).toList == List(2, 3, 5, 7, 11, 13, 17, 19, 23, 29))

// Previously calculated values were memoized, as shown by toString
assert(primes.toString == "Stream(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, ?)")
```

Lea Corrientes en línea: https://riptutorial.com/es/scala/topic/3702/corrientes

Capítulo 15: Cuasiquotes

Examples

Crear un árbol de sintaxis con quasiquotes

Usa cuasicotas para crear un `Tree` en una macro.

```
object macro {
  def addCreationDate(): java.util.Date = macro impl.addCreationDate
}

object impl {
  def addCreationDate(c: Context)(): c.Expr[java.util.Date] = {
    import c.universe._

    val date = q"new java.util.Date()" // this is the quasiquote
    c.Expr[java.util.Date](date)
  }
}
```

Puede ser arbitrariamente complejo, pero será validado para la sintaxis de scala correcta.

Lea Cuasiquotes en línea: https://riptutorial.com/es/scala/topic/4032/cuasiquotes

Capítulo 16: Enumeraciones

Observaciones

Se prefiere el enfoque con `sealed trait` y `case objects` porque la enumeración de Scala tiene algunos problemas:

1. Las enumeraciones tienen el mismo tipo después del borrado.
2. El compilador no se queja sobre "La coincidencia no es exhaustiva", si se pierde el caso, fallará en el tiempo de ejecución de `scala.MatchError` :

```
def isWeekendWithBug(day: WeekDays.Value): Boolean = day match {
  case WeekDays.Sun | WeekDays.Sat => true
}

isWeekendWithBug(WeekDays.Fri)
scala.MatchError: Fri (of class scala.Enumeration$Val)
```

Comparar con:

```
def isWeekendWithBug(day: WeekDay): Boolean = day match {
  case WeekDay.Sun | WeekDay.Sat => true
}

Warning: match may not be exhaustive.
It would fail on the following inputs: Fri, Mon, Thu, Tue, Wed
def isWeekendWithBug(day: WeekDay): Boolean = day match {
                                              ^
```

Una explicación más detallada se presenta en este artículo sobre Scala Enumeration .

Examples

Días de la semana usando Scala Enumeration

Las enumeraciones similares a Java se pueden crear extendiendo la enumeración .

```
object WeekDays extends Enumeration {
  val Mon, Tue, Wed, Thu, Fri, Sat, Sun = Value
}

def isWeekend(day: WeekDays.Value): Boolean = day match {
  case WeekDays.Sat | WeekDays.Sun => true
  case _ => false
}

isWeekend(WeekDays.Sun)
res0: Boolean = true
```

También es posible agregar un nombre legible para los valores en una enumeración:

```
object WeekDays extends Enumeration {
      val Mon = Value("Monday")
      val Tue = Value("Tuesday")
      val Wed = Value("Wednesday")
      val Thu = Value("Thursday")
      val Fri = Value("Friday")
      val Sat = Value("Saturday")
      val Sun = Value("Sunday")
}

println(WeekDays.Mon)
>> Monday

WeekDays.withName("Monday") == WeekDays.Mon
>> res0: Boolean = true
```

Tenga cuidado con el comportamiento no tan seguro de los tipos, en el que diferentes enumeraciones se pueden evaluar como el mismo tipo de instancia:

```
object Parity extends Enumeration {
   val Even, Odd = Value
}

WeekDays.Mon.isInstanceOf[Parity.Value]
>> res1: Boolean = true
```

Usando rasgos sellados y objetos de caja

Una alternativa a la extensión de la `Enumeration` es usar objetos de casos `sealed` :

```
sealed trait WeekDay

object WeekDay {
  case object Mon extends WeekDay
  case object Tue extends WeekDay
  case object Wed extends WeekDay
  case object Thu extends WeekDay
  case object Fri extends WeekDay
  case object Sun extends WeekDay
  case object Sat extends WeekDay
}
```

La palabra clave `sealed` garantiza que el rasgo `WeekDay` no se puede extender en otro archivo. Esto le permite al compilador hacer ciertas suposiciones, incluyendo que todos los valores posibles de `WeekDay` ya están enumerados.

Un inconveniente es que este método no le permite obtener una lista de todos los valores posibles. Para obtener dicha lista se debe proporcionar explícitamente:

```
val allWeekDays = Seq(Mon, Tue, Wed, Thu, Fri, Sun, Sat)
```

Las clases de casos también pueden extender un rasgo `sealed` . Por lo tanto, los objetos y las clases de casos se pueden mezclar para crear jerarquías complejas:

```
sealed trait CelestialBody

object CelestialBody {
  case object Earth extends CelestialBody
  case object Sun extends CelestialBody
  case object Moon extends CelestialBody
  case class Asteroid(name: String) extends CelestialBody
}
```

Otro inconveniente es que no hay forma de acceder al nombre de variable de la enumeración de un objeto `sealed` , o buscar por él. Si necesita algún tipo de nombre asociado a cada valor, debe definirse manualmente:

```
  sealed trait WeekDay { val name: String }

  object WeekDay {
      case object Mon extends WeekDay { val name = "Monday" }
      case object Tue extends WeekDay { val name = "Tuesday" }
      (...)
  }
```

O solo:

```
  sealed case class WeekDay(name: String)

  object WeekDay {
      object Mon extends WeekDay("Monday")
      object Tue extends WeekDay("Tuesday")
      (...)
  }
```

Uso de rasgos sellados y objetos de caja y allValues-macro

Esta es solo una extensión de la variante de rasgo sellado donde una macro genera un conjunto con todas las instancias en tiempo de compilación. Esto omite el inconveniente de que un desarrollador puede agregar un valor a la enumeración, pero se olvida de agregarlo al conjunto allElements.

Esta variante es especialmente útil para grandes enumeraciones.

```
import EnumerationMacros._

sealed trait WeekDay
object WeekDay {
  case object Mon extends WeekDay
  case object Tue extends WeekDay
  case object Wed extends WeekDay
  case object Thu extends WeekDay
  case object Fri extends WeekDay
  case object Sun extends WeekDay
  case object Sat extends WeekDay
  val allWeekDays: Set[WeekDay] = sealedInstancesOf[WeekDay]
}
```

Para que esto funcione necesitas esta macro:

```
import scala.collection.immutable.TreeSet
import scala.language.experimental.macros
import scala.reflect.macros.blackbox

/**
A macro to produce a TreeSet of all instances of a sealed trait.
Based on Travis Brown's work:
http://stackoverflow.com/questions/13671734/iteration-over-a-sealed-trait-in-scala
CAREFUL: !!! MUST be used at END OF code block containing the instances !!!
*/
object EnumerationMacros {
  def sealedInstancesOf[A]: TreeSet[A] = macro sealedInstancesOf_impl[A]

  def sealedInstancesOf_impl[A: c.WeakTypeTag](c: blackbox.Context) = {
    import c.universe._

    val symbol = weakTypeOf[A].typeSymbol.asClass

    if  (!symbol.isClass || !symbol.isSealed)
      c.abort(c.enclosingPosition, "Can only enumerate values of a sealed trait or class.")
    else {

      val children = symbol.knownDirectSubclasses.toList

      if (!children.forall(_.isModuleClass)) c.abort(c.enclosingPosition, "All children must
be objects.")
      else c.Expr[TreeSet[A]] {

        def sourceModuleRef(sym: Symbol) =
Ident(sym.asInstanceOf[scala.reflect.internal.Symbols#Symbol
          ].sourceModule.asInstanceOf[Symbol]
        )

        Apply(
          Select(
            reify(TreeSet).tree,
            TermName("apply")
          ),
          children.map(sourceModuleRef(_))
        )
      }
    }
  }
}
```

Lea Enumeraciones en línea: https://riptutorial.com/es/scala/topic/1499/enumeraciones

Capítulo 17: Expresiones regulares

Sintaxis

- re.findAllIn (s: CharSequence): MatchIterator
- re.findAllMatchIn (s: CharSequence): Iterator [Match]
- re.findFirstIn (s: CharSequence): Option [String]
- re.findFirstMatchIn (s: CharSequence): Option [Match]
- re.findPrefixMatchIn (s: CharSequence): Option [Match]
- re.findPrefixOf (s: CharSequence): Option [String]
- re.replaceAllIn (s: CharSequence, reemplazador: Match => String): String
- re.replaceAllIn (s: CharSequence, reemplazo: String): String
- re.replaceFirstIn (s: CharSequence, reemplazo: String): String
- re.replaceSomeIn (s: CharSequence, reemplazador: Match => Option [String]): String
- re.split (s: CharSequence): Array [String]

Examples

Declarar expresiones regulares

El método `r` provisto implícitamente a través de scala.collection.immutable.StringOps produce una instancia de scala.util.matching.Regex de la cadena del sujeto. La sintaxis de cadena entre comillas triples de Scala es útil aquí, ya que no tiene que escapar de las barras invertidas como lo haría en Java:

```
val r0: Regex = """(\d{4})-(\d${2})-(\d{2})""".r     // :)
val r1: Regex = "(\\d{4})-(\\d{2})-(\\d{2})".r // :(
```

scala.util.matching.Regex implementa una API de expresión regular idiomática para Scala como un contenedor sobre java.util.regex.Pattern , y la sintaxis compatible es la misma. Dicho esto, el soporte de Scala para los literales de cadenas de varias líneas hace que la marca `x` mucho más útil, permitiendo comentarios e ignorando espacios en blanco de patrones:

```
val dateRegex = """(?x:
   (\d{4}) # year
  -(\d{2}) # month
  -(\d{2}) # day
)""".r
```

Hay una versión sobrecargada de `r` , `def r(names: String*): Regex` que le permite asignar nombres de grupo a sus capturas de patrones. Esto es algo frágil ya que los nombres están disociados de las capturas, y solo deben usarse si la expresión regular se usará en múltiples ubicaciones:

```
"""(\d{4})-(\d{2})-(\d{2})""".r("y", "m", "d").findFirstMatchIn(str) match {
  case Some(matched) =>
    val y = matched.group("y").toInt
```

```
    val m = matched.group("m").toInt
    val d = matched.group("d").toInt
    java.time.LocalDate.of(y, m, d)
  case None => ???
}
```

Repetir la coincidencia de un patrón en una cadena

```
val re = """\((.*?)\)""".r

val str =
"(The)(example)(of)(repeating)(pattern)(in)(a)(single)(string)(I)(had)(some)(trouble)(with)(once)"


re.findAllMatchIn(str).map(_.group(1)).toList
res2: List[String] = List(The, example, of, repeating, pattern, in, a, single, string, I, had,
some, trouble, with, once)
```

Lea Expresiones regulares en línea: https://riptutorial.com/es/scala/topic/2891/expresiones-regulares

Capítulo 18: Extractores

Sintaxis

- extractor de valores (extractValue1, _ / * segundo valor extraído ignorado * /) = valueToBeExtracted
- valueToBeExtracted match {case extractor (extractValue1, _) => ???}
- val (tuple1, tuple2, tuple3) = tupleWith3Elements
- objeto Foo {def unapply (foo: Foo): Opción [String] = Some (foo.x); }

Examples

Extractores de tuplas

`x` e `y` se extraen de la tupla:

```
val (x, y) = (1337, 42)
// x: Int = 1337
// y: Int = 42
```

Para ignorar un valor usa _ :

```
val (_, y: Int) = (1337, 42)
// y: Int = 42
```

Para desembalar un extractor:

```
val myTuple = (1337, 42)
myTuple._1  // res0: Int = 1337
myTuple._2  // res1: Int = 42
```

Tenga en cuenta que las tuplas tienen una longitud máxima de 22 y, por lo tanto, de `._1` a `._22` funcionarán (suponiendo que la tupla tenga al menos ese tamaño).

Los extractores de tuplas se pueden usar para proporcionar argumentos simbólicos para funciones literales:

```
val persons = List("A." -> "Lovelace", "G." -> "Hopper")
val names = List("Lovelace, A.", "Hopper, G.")

assert {
  names ==
    (persons map { name =>
      s"${name._2}, ${name._1}"
    })
}

assert {
```

```
  names ==
    (persons map { case (given, surname) =>
      s"$surname, $given"
    })
}
```

Case Class Extractors

Una clase de caso es una clase con una gran cantidad de código estándar incluido automáticamente. Una ventaja de esto es que Scala facilita el uso de extractores con clases de casos.

```
case class Person(name: String, age: Int)  // Define the case class
val p = Person("Paola", 42)  // Instantiate a value with the case class type

val Person(n, a) = p  // Extract values n and a
// n: String = Paola
// a: Int = 42
```

En esta coyuntura, tanto `n` como `a` son `val` en el programa y se puede acceder a ellos como tales: se dice que se han "extraído" de la pág. Continuo:

```
val p2 = Person("Angela", 1337)

val List(Person(n1, a1), Person(_, a2)) = List(p, p2)
// n1: String = Paola
// a1: Int = 42
// a2: Int = 1337
```

Aquí vemos dos cosas importantes:

- La extracción puede ocurrir en niveles "profundos": se pueden extraer las propiedades de los objetos anidados.
- No todos los elementos *necesitan* ser extraídos. El carácter comodín _ indica que ese valor en particular puede ser cualquier cosa, y se ignora. No se crea ningún `val` .

En particular, esto puede facilitar la comparación entre colecciones:

```
val ls = List(p1, p2, p3)  // List of Person objects
ls.map(person => person match {
  case Person(n, a) => println("%s is %d years old".format(n, a))
})
```

Aquí, tenemos un código que utiliza el extractor para verificar explícitamente que la `person` es un objeto `Person` y que saca de inmediato las variables que nos interesan: `n` y `a` .

Unapply - Extractores personalizados

Una extracción personalizado puede ser escrito por la aplicación de la `unapply` método y devolver un valor de tipo `Option` :

```
class Foo(val x: String)

object Foo {
  def unapply(foo: Foo): Option[String] = Some(foo.x)
}

new Foo("42") match {
  case Foo(x) => x
}
// "42"
```

El tipo de `unapply` de `unapply` puede ser diferente a la `Option` , siempre que el tipo devuelto proporcione los métodos `get` e `isEmpty` . En este ejemplo, la `Bar` se define con esos métodos y la `unapply` devuelve una instancia de la `Bar` :

```
class Bar(val x: String) {
  def get = x
  def isEmpty = false
}

object Bar {
  def unapply(bar: Bar): Bar = bar
}

new Bar("1337") match {
  case Bar(x) => x
}
// "1337"
```

El tipo de `unapply` de `unapply` también puede ser un `Boolean` , que es un caso especial que no cumple con los requisitos `get` y `isEmpty` anteriores. Sin embargo, tenga en cuenta en este ejemplo que `DivisibleByTwo` es un objeto, no una clase, y no toma un parámetro (y, por lo tanto, ese parámetro no puede ser enlazado):

```
object DivisibleByTwo {
  def unapply(num: Int): Boolean = num % 2 == 0
}

4 match {
  case DivisibleByTwo() => "yes"
  case _ => "no"
}
// yes

3 match {
  case DivisibleByTwo() => "yes"
  case _ => "no"
}
// no
```

Recuerde que la `unapply` va en el objeto compañero de una clase, no en la clase. El ejemplo anterior será claro si entiende esta distinción.

Extractor de notación infijo.

Si una clase de caso tiene exactamente dos valores, su extractor puede usarse en notación de infijo.

```
case class Pair(a: String, b: String)
val p: Pair = Pair("hello", "world")
val x Pair y = p
//x: String = hello
//y: String = world
```

Cualquier extractor que devuelva un 2-tuple puede funcionar de esta manera.

```
object Foo {
    def unapply(s: String): Option[(Int, Int)] = Some((s.length, 5))
}
val a Foo b = "hello world!"
//a: Int = 12
//b: Int = 5
```

Extractores Regex

Una expresión regular con partes agrupadas se puede utilizar como un extractor:

```
scala> val address = """(.+):(\d+)""".r
address: scala.util.matching.Regex = (.+):(\d+)

scala> val address(host, port) = "some.domain.org:8080"
host: String = some.domain.org
port: String = 8080
```

Tenga en cuenta que cuando no `MatchError` se `MatchError` un `MatchError` en tiempo de ejecución:

```
scala> val address(host, port) = "something not a host and port"
scala.MatchError: something not a host and port (of class java.lang.String)
```

Extractores transformadores

El comportamiento del extractor se puede usar para derivar valores arbitrarios de su entrada. Esto puede ser útil en situaciones en las que desea poder actuar sobre los resultados de una transformación en caso de que la transformación sea exitosa.

Considere como ejemplo los diversos formatos de nombre de usuario que se pueden usar en un entorno Windows :

```
object UserPrincipalName {
  def unapply(str: String): Option[(String, String)] = str.split('@') match {
    case Array(u, d) if u.length > 0 && d.length > 0 => Some((u, d))
    case _ => None
  }
}

object DownLevelLogonName {
  def unapply(str: String): Option[(String, String)] = str.split('\\') match {
```

```
    case Array(d, u) if u.length > 0 && d.length > 0 => Some((d, u))
    case _ => None
  }
}

def getDomain(str: String): Option[String] = str match {
  case UserPrincipalName(_, domain) => Some(domain)
  case DownLevelLogonName(domain, _) => Some(domain)
  case _ => None
}
```

De hecho, es posible crear un extractor que muestre ambos comportamientos ampliando los tipos que puede igualar:

```
object UserPrincipalName {
  def unapply(obj: Any): Option[(String, String)] = obj match {
    case upn: UserPrincipalName => Some((upn.username, upn.domain))
    case str: String => str.split('@') match {
      case Array(u, d) if u.length > 0 && d.length > 0 => Some((u, d))
      case _ => None
    }
    case _ => None
  }
}
```

En general, los extractores son simplemente una reformulación conveniente del patrón de `Option` , como se aplica a métodos con nombres como `tryParse` :

```
UserPrincipalName.unapply("user@domain") match {
  case Some((u, d)) => ???
  case None => ???
}
```

Lea Extractores en línea: https://riptutorial.com/es/scala/topic/930/extractores

Capítulo 19: Función de orden superior

Observaciones

Scala hace todo lo posible para tratar los métodos y las funciones como sintácticamente idénticos. Pero bajo el capó, son conceptos distintos.

Un método es un código ejecutable y no tiene representación de valor.

Una función es una instancia de objeto real de tipo `Function1` (o un tipo similar de otra aridad). Su código está contenido en su método de `apply` . Efectivamente, simplemente actúa como un valor que puede transmitirse.

Por cierto, la capacidad de tratar las funciones como valores *es* exactamente lo que se entiende por un lenguaje que tiene soporte para funciones de orden superior. Las instancias de funciones son el enfoque de Scala para implementar esta característica.

Una función de orden superior real es una función que toma un valor de función como argumento o devuelve un valor de función. Pero en Scala, como todas las operaciones son métodos, es más general pensar en métodos que reciben o devuelven parámetros de función. Entonces, el `map` , tal como se define en `Seq` podría considerarse como una "función de orden superior" debido a que su parámetro es una función, pero no es literalmente una función; es un metodo

Examples

Usando métodos como valores de función

El compilador Scala convertirá automáticamente los métodos en valores de función con el fin de pasarlos a funciones de orden superior.

```
object MyObject {
  def mapMethod(input: Int): String = {
    int.toString
  }
}

Seq(1, 2, 3).map(MyObject.mapMethod) // Seq("1", "2", "3")
```

En el ejemplo anterior, `MyObject.mapMethod` no es una llamada de función, sino que se pasa a `map` como un valor. De hecho, el `map` *requiere que se le* pase un valor de función, como se puede ver en su firma. La firma para el `map` de una `List[A]` (una lista de objetos de tipo `A`) es:

```
def map[B](f: (A) ⇒ B): List[B]
```

La parte `f: (A) => B` indica que el parámetro para esta llamada de método es alguna función que toma un objeto de tipo `A` y devuelve un objeto de tipo `B` `A` y `B` se definen arbitrariamente. Volviendo al primer ejemplo, podemos ver que `mapMethod` toma un `Int` (que corresponde a `A`) y devuelve una

`String` (que corresponde a `B`). Por `mapMethod` tanto, `mapMethod` es un valor de función válido para pasar al `map` . Podríamos reescribir el mismo código así:

```
Seq(1, 2, 3).map(x:Int => int.toString)
```

Esto incluye el valor de la función, que puede agregar claridad a las funciones simples.

Funciones de alto orden (función como parámetro)

Una función de orden superior, a diferencia de una función de primer orden, puede tener una de tres formas:

- Uno o más de sus parámetros es una función y devuelve algún valor.
- Devuelve una función, pero ninguno de sus parámetros es una función.
- Ambos de los anteriores: uno o más de sus parámetros es una función, y devuelve una función.

```
object HOF {
    def main(args: Array[String]) {
    val list =
List(("Srini","E"),("Subash","R"),("Ranjith","RK"),("Vicky","s"),("Sudhar","s"))
    //HOF
     val fullNameList= list.map(n => getFullName(n._1, n._2))

     }

    def getFullName(firstName: String, lastName: String): String = firstName + "." +
lastName
    }
```

Aquí la función de mapa toma una función `getFullName(n._1,n._2)` como parámetro. Esto se llama **HOF (función de orden superior).**

Argumentos perezosos de evaluación

Scala admite la evaluación perezosa para argumentos de funciones usando notación: `def func(arg: => String)` . Dicho argumento de función podría tomar un objeto `String` normal o una función de orden superior con `String` tipo de retorno `String` . En el segundo caso, el argumento de la función sería evaluado en el acceso al valor.

Por favor vea el ejemplo:

```
def calculateData: String = {
  print("Calculating expensive data! ")
  "some expensive data"
}

def dumbMediator(preconditions: Boolean, data: String): Option[String] = {
  print("Applying mediator")
  preconditions match {
```

```
    case true => Some(data)
    case false => None
  }
}

def smartMediator(preconditions: Boolean, data: => String): Option[String] = {
  print("Applying mediator")
  preconditions match {
    case true => Some(data)
    case false => None
  }
}

smartMediator(preconditions = false, calculateData)

dumbMediator(preconditions = false, calculateData)
```

`smartMediator` llamada a `smartMediator` devolverá el valor Ninguno e imprimirá el mensaje `"Applying mediator"` .

`dumbMediator` llamada a `dumbMediator` devolverá el valor Ninguno e imprimirá el mensaje `"Calculating expensive data! Applying mediator"` .

La evaluación perezosa puede ser extremadamente útil cuando desea optimizar una sobrecarga de cálculo de argumentos caros.

Lea Función de orden superior en línea: https://riptutorial.com/es/scala/topic/1642/funcion-de-orden-superior

Capítulo 20: Funciones

Observaciones

Scala tiene funciones de primera clase.

Diferencia entre funciones y métodos:

Una función no es un método en Scala: las funciones son un valor y pueden asignarse como tales. Los métodos (creados usando `def`), por otro lado, deben pertenecer a una clase, rasgo u objeto.

- Las funciones se compilan en una clase que extiende un rasgo (como `Function1`) en tiempo de compilación, y se crean instancias a un valor en tiempo de ejecución. Los métodos, por otro lado, son miembros de su clase, rasgo u objeto, y no existen fuera de eso.
- Un método se puede convertir en una función, pero una función no se puede convertir en un método.
- Los métodos pueden tener parametrización de tipo, mientras que las funciones no.
- Los métodos pueden tener valores predeterminados de parámetros, mientras que las funciones no pueden.

Examples

Funciones anónimas

Las funciones anónimas son funciones que están definidas pero no se les asigna un nombre.

La siguiente es una función anónima que toma dos enteros y devuelve la suma.

```
(x: Int, y: Int) => x + y
```

La expresión resultante se puede asignar a un `val` :

```
val sum = (x: Int, y: Int) => x + y
```

Las funciones anónimas se utilizan principalmente como argumentos para otras funciones. Por ejemplo, la función de `map` en una colección espera otra función como argumento:

```
// Returns Seq("FOO", "BAR", "QUX")
Seq("Foo", "Bar", "Qux").map((x: String) => x.toUpperCase)
```

Los tipos de los argumentos de la función anónima se pueden omitir: los tipos se deducen automáticamente :

```
Seq("Foo", "Bar", "Qux").map((x) => x.toUpperCase)
```

Si hay un solo argumento, se pueden omitir los paréntesis alrededor de ese argumento:

```
Seq("Foo", "Bar", "Qux").map(x => x.toUpperCase)
```

Subraya la taquigrafía

Hay una sintaxis aún más corta que no requiere nombres para los argumentos. El fragmento anterior se puede escribir:

```
Seq("Foo", "Bar", "Qux").map(_.toUpperCase)
```

_ representa los argumentos de la función anónima en posición. Con una función anónima que tiene múltiples parámetros, cada aparición de _ se referirá a un argumento diferente. Por ejemplo, las dos expresiones siguientes son equivalentes:

```
// Returns "FooBarQux" in both cases
Seq("Foo", "Bar", "Qux").reduce((s1, s2) => s1 + s2)
Seq("Foo", "Bar", "Qux").reduce(_ + _)
```

Al usar esta abreviatura, cualquier argumento representado por la posición _ solo puede ser referenciado una sola vez y en el mismo orden.

Funciones anónimas sin parámetros

Para crear un valor para una función anónima que no toma parámetros, deje la lista de parámetros en blanco:

```
val sayHello = () => println("hello")
```

Composición

La composición de funciones permite que dos funciones funcionen y se vean como una sola función. Expresado en términos matemáticos, dada una función `f(x)` y una función `g(x)` , la función `h(x) = f(g(x))` .

Cuando se compila una función, se compila a un tipo relacionado con `Function1` . Scala proporciona dos métodos en la `Function1` aplicación relacionada a la composición: `andThen` y `compose` . El método de `compose` encaja con la definición matemática anterior así:

```
val f: B => C = ...
val g: A => B = ...

val h: A => C = f compose g
```

El `andThen` (piensa que `h(x) = g(f(x))`) tiene una sensación más parecida a DSL:

```
val f: A => B = ...
val g: B => C = ...

val h: A => C = f andThen g
```

Se asigna una nueva función anónima que se cierra sobre `f` y `g` . Esta función está vinculada a la nueva función `h` en ambos casos.

```
def andThen(g: B => C): A => C = new (A => C){
  def apply(x: A) = g(self(x))
}
```

Si cualquiera de `f` o `g` funciona a través de un efecto secundario, llamar a `h` hará que todos los efectos secundarios de `f` y `g` ocurran en el pedido. Lo mismo ocurre con cualquier cambio de estado mutable.

Relación a funciones parciales

```
trait PartialFunction[-A, +B] extends (A => B)
```

Cada-solo argumento `PartialFunction` es también un `Function1` . Esto es contraintuitivo en un sentido matemático formal, pero se ajusta mejor al diseño orientado a objetos. Por esta razón, `Function1` no tiene que proporcionar un método `true isDefinedAt` constante.

Para definir una función parcial (que también es una función), use la siguiente sintaxis:

```
{ case i: Int => i + 1 } // or equivalently { case i: Int ⇒ i + 1 }
```

Para más detalles, eche un vistazo a PartialFunctions .

Lea Funciones en línea: https://riptutorial.com/es/scala/topic/477/funciones

Capítulo 21: Funciones definidas por el usuario para Hive

Examples

Un simple UDF Hive dentro de Apache Spark

```
import org.apache.spark.sql.functions._

// Create a function that uses the content of the column inside the dataframe
val code = (param: String) => if (param == "myCode") 1 else 0
// With that function, create the udf function
val myUDF = udf(code)
// Apply the udf to a column inside the existing dataframe, creating a dataframe with the
additional new column
val newDataframe = aDataframe.withColumn("new_column_name", myUDF(col(inputColumn)))
```

Lea Funciones definidas por el usuario para Hive en línea:
https://riptutorial.com/es/scala/topic/8241/funciones-definidas-por-el-usuario-para-hive

Capítulo 22: Funciones parciales

Examples

Composición

Las funciones parciales se usan a menudo para definir una función total en partes:

```
sealed trait SuperType
case object A extends SuperType
case object B extends SuperType
case object C extends SuperType

val pfA: PartialFunction[SuperType, Int] = {
  case A => 5
}

val pfB: PartialFunction[SuperType, Int] = {
  case B => 10
}

val input: Seq[SuperType] = Seq(A, B, C)

input.map(pfA orElse pfB orElse {
  case _ => 15
}) // Seq(5, 10, 15)
```

En este uso, las funciones parciales se intentan en orden de concatenación con el método `orElse` . Normalmente, se proporciona una función parcial final que coincide con todos los casos restantes. En conjunto, la combinación de estas funciones actúa como una función total.

Este patrón se usa normalmente para separar las preocupaciones cuando una función puede actuar efectivamente como un despachador para rutas de código dispares. Esto es común, por ejemplo, en el método de recepción de un Actor Akka .

Uso con `collect`

Si bien la función parcial se usa a menudo como una sintaxis conveniente para funciones totales, al incluir una coincidencia de comodín final (`case _`), en algunos métodos, su parcialidad es clave. Un ejemplo muy común en Scala idiomático es el método de `collect` , definido en la biblioteca de colecciones de Scala. Aquí, las funciones parciales permiten que las funciones comunes de examinar los elementos de una colección para mapearlos y / o filtrarlos ocurran en una sintaxis compacta.

Ejemplo 1

Suponiendo que tenemos una función de raíz cuadrada definida como función parcial:

```
val sqRoot:PartialFunction[Double,Double] = { case n if n > 0 => math.sqrt(n) }
```

Podemos invocarlo con el combinador de `collect` :

```
List(-1.1,2.2,3.3,0).collect(sqRoot)
```

Realizando efectivamente la misma operación que:

```
List(-1.1,2.2,3.3,0).filter(sqRoot.isDefinedAt).map(sqRoot)
```

Ejemplo 2

```
sealed trait SuperType // `sealed` modifier allows inheritance within current build-unit only
case class A(value: Int) extends SuperType
case class B(text: String) extends SuperType
case object C extends SuperType

val input: Seq[SuperType] = Seq(A(5), B("hello"), C, A(25), B(""))

input.collect {
  case A(value) if value < 10   => value.toString
  case B(text) if text.nonEmpty => text
} // Seq("5", "hello")
```

Hay varias cosas a tener en cuenta en el ejemplo anterior:

- El lado izquierdo de cada coincidencia de patrón selecciona efectivamente los elementos para procesar e incluir en la salida. Cualquier valor que no tenga un `case` coincidente simplemente se omite.
- El lado derecho define el procesamiento específico de caso para aplicar.
- La coincidencia de patrones se une a la variable para su uso en declaraciones de guarda (las cláusulas `if`) y el lado derecho.

Sintaxis basica

Scala tiene un tipo especial de función llamada función parcial , que se extiende a las funciones normales , lo que significa que una instancia de `PartialFunction` se puede usar donde se espera que `Function1` . Las funciones parciales se pueden definir de forma anónima utilizando la sintaxis de `case` también se usa en la coincidencia de patrones :

```
val pf: PartialFunction[Boolean, Int] = {
  case true => 7
}

pf.isDefinedAt(true) // returns true
pf(true) // returns 7

pf.isDefinedAt(false) // returns false
pf(false) // throws scala.MatchError: false (of class java.lang.Boolean)
```

Como se ve en el ejemplo, una función parcial no necesita definirse en todo el dominio de su primer parámetro. Se supone que una instancia de `Function1` estándar es *total* , lo que significa que se define para cada argumento posible.

Uso como una función total

Las funciones parciales son muy comunes en Scala idiomático. A menudo se usan por su conveniente sintaxis basada en `case` para definir funciones totales sobre rasgos :

```
sealed trait SuperType // `sealed` modifier allows inheritance within current build-unit only
case object A extends SuperType
case object B extends SuperType
case object C extends SuperType

val input: Seq[SuperType] = Seq(A, B, C)

input.map {
  case A => 5
  case _ => 10
} // Seq(5, 10, 10)
```

Esto guarda la sintaxis adicional de una declaración de `match` en una función anónima regular. Comparar:

```
input.map { item =>
  item match {
    case A => 5
    case _ => 10
  }
} // Seq(5, 10, 10)
```

También se utiliza con frecuencia para realizar una descomposición de parámetros utilizando la coincidencia de patrones, cuando se pasa una tupla o una clase de caso a una función:

```
val input = Seq("A" -> 1, "B" -> 2, "C" -> 3)

input.map { case (a, i) =>
   a + i.toString
} // Seq("A1", "B2", "C3")
```

Uso para extraer tuplas en una función de mapa

Estas tres funciones de mapa son equivalentes, por lo tanto, use la variación que su equipo encuentre más legible.

```
val numberNames = Map(1 -> "One", 2 -> "Two", 3 -> "Three")

// 1. No extraction
numberNames.map(it => s"${it._1} is written ${it._2}" )

// 2. Extraction within a normal function
numberNames.map(it => {
    val (number, name) = it
    s"$number is written $name"
})

// 3. Extraction via a partial function (note the brackets in the parentheses)
numberNames.map({ case (number, name) => s"$number is written $name" })
```

La función parcial **debe coincidir con todas las entradas** : cualquier caso que no coincida producirá una excepción en el tiempo de ejecución.

Lea Funciones parciales en línea: https://riptutorial.com/es/scala/topic/1638/funciones-parciales

Capítulo 23: Futuros

Examples

Creando un futuro

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

object FutureDivider {
    def divide(a: Int, b: Int): Future[Int] = Future {
        // Note that this is integer division.
        a / b
    }
}
```

Simplemente, el método de `divide` crea un Futuro que se resolverá con el cociente de `a` sobre `b` .

Consumiendo un futuro exitoso

La forma más fácil de consumir un futuro *exitoso,* o más bien, obtener el valor dentro del futuro, es usar el método de `map` . Supongamos que un cierto código llama a la `divide` método de la `FutureDivider` objeto del ejemplo "Creando un futuro". ¿Cómo debería ser el código para obtener el cociente de `a` sobre `b` ?

```
object Calculator {
    def calculateAndReport(a: Int, b: Int) = {
        val eventualQuotient = FutureDivider divide(a, b)

        eventualQuotient map {
            quotient => println(quotient)
        }
    }
}
```

Consumiendo un futuro fallido

A veces, el cálculo en un futuro puede crear una excepción, lo que hará que el futuro falle. En el ejemplo de "Creación de un futuro", ¿qué sucede si el código de llamada pasó `55` y `0` al método de `divide` ? Lanzaría una `ArithmeticException` después de tratar de dividir por cero, por supuesto. ¿Cómo se manejaría eso en consumir código? En realidad, hay un puñado de maneras de lidiar con los fracasos.

Manejar la excepción con `recover` y coincidencia de patrones.

```
object Calculator {
    def calculateAndReport(a: Int, b: Int) = {
        val eventualQuotient = FutureDivider divide(a, b)
```

```
        eventualQuotient recover {
            case ex: ArithmeticException => println(s"It failed with: ${ex.getMessage}")
        }
    }
}
```

Maneje la excepción con la proyección `failed` , donde la excepción se convierte en el valor del futuro:

```
object Calculator {
    def calculateAndReport(a: Int, b: Int) = {
        val eventualQuotient = FutureDivider divide(a, b)

        // Note the use of the dot operator to get the failed projection and map it.
        eventualQuotient.failed.map {
            ex => println(s"It failed with: ${ex.getMessage}")
        }
    }
}
```

Poniendo el futuro juntos

Los ejemplos anteriores demostraron las características individuales de un futuro, manejando casos de éxito y fracaso. Generalmente, sin embargo, ambas características se manejan mucho más tersamente. Aquí está el ejemplo, escrito de una manera más ordenada y más realista:

```
object Calculator {
    def calculateAndReport(a: Int, b: Int) = {
        val eventualQuotient = FutureDivider divide(a, b)

        eventualQuotient map {
            quotient => println(s"Quotient: $quotient")
        } recover {
            case ex: ArithmeticException => println(s"It failed with: ${ex.getMessage}")
        }
    }
}
```

Secuenciación y travesía de futuros.

En algunos casos es necesario calcular una cantidad variable de valores en futuros separados. Suponga que tiene una `List[Future[Int]]` , pero en su lugar debe procesarse una `List[Int]` . Entonces la pregunta es cómo convertir esta instancia de la `List[Future[Int]]` en un `Future[List[Int]]` . Para este propósito existe el método de `sequence` en el objeto compañero `Future` .

```
def listOfFuture: List[Future[Int]] = List(1,2,3).map(Future(_))
def futureOfList: Future[List[Int]] = Future.sequence(listOfFuture)
```

En general, la `sequence` es un operador comúnmente conocido dentro del mundo de la programación funcional que transforma `F[G[T]]` en `G[F[T]]` con restricciones a `F` y `G`

Hay un operador alternativo llamado `traverse` , que funciona de manera similar pero toma una función como un argumento adicional. Con la función de identidad `x => x` como parámetro, se comporta como el operador de `sequence` .

```
def listOfFuture: List[Future[Int]] = List(1,2,3).map(Future(_))
def futureOfList: Future[List[Int]] = Future.traverse(listOfFuture)(x => x)
```

Sin embargo, el argumento adicional permite modificar cada instancia futura dentro del `listOfFuture` dado. Además, el primer argumento no necesita ser una lista de `Future` . Por lo tanto es posible transformar el ejemplo de la siguiente manera:

```
def futureOfList: Future[List[Int]] = Future.traverse(List(1,2,3))(Future(_))
```

En este caso, la `List(1,2,3)` se pasa directamente como primer argumento y la función de identidad `x => x` se reemplaza con la función `Future(_)` para envolver de manera similar cada valor de `Int` en un `Future` . Una ventaja de esto es que la `List[Future[Int]]` intermedia `List[Future[Int]]` puede omitirse para mejorar el rendimiento.

Combina Futuros Múltiples - Para Comprensión

La *comprensión* es una forma compacta de ejecutar un bloque de código que depende del resultado exitoso de múltiples futuros.

Con `f1, f2, f3` tres `Future[String]` que contendrán las cadenas `one, two, three` respectivamente,

```
val fCombined =
    for {
        s1 <- f1
        s2 <- f2
        s3 <- f3
    } yield (s"$s1 - $s2 - $s3")
```

`fCombined` será un `Future[String]` contiene la cadena `one - two - three` una vez que todos los futuros se hayan completado con éxito.

Tenga en cuenta que aquí se supone un ExectionContext implícito.

Además, tenga en cuenta que para la comprensión es solo un azúcar sintáctico para un método flatMap, por lo que la construcción de objetos Futuros dentro del cuerpo eliminaría la ejecución concurrente de bloques de código incluidos en futuros y llevaría a un código secuencial. Lo ves en el ejemplo:

```
val result1 = for {
  first <- Future {
    Thread.sleep(2000)
    System.currentTimeMillis()
  }
  second <- Future {
    Thread.sleep(1000)
    System.currentTimeMillis()
```

```
  }
} yield first - second

val fut1 = Future {
  Thread.sleep(2000)
  System.currentTimeMillis()
}
val fut2 = Future {
  Thread.sleep(1000)
  System.currentTimeMillis()
}
val result2 = for {
  first <- fut1
  second <- fut2
} yield first - second
```

Valor encerrada por `result1` objeto sería siempre negativo, mientras que `result2` serían positivos.

Para obtener más detalles sobre la *comprensión* y el `yield` en general, consulte http://docs.scala-lang.org/tutorials/FAQ/yield.html

Lea Futuros en línea: https://riptutorial.com/es/scala/topic/3245/futuros

Capítulo 24: Implícitos

Sintaxis

- val implícito x: T = ???

Observaciones

Las clases implícitas permiten que se agreguen métodos personalizados a los tipos existentes, sin tener que modificar su código, enriqueciendo así los tipos sin necesidad de controlar el código.

El uso de tipos implícitos para enriquecer una clase existente a menudo se denomina patrón de "enriquecer mi biblioteca".

Restricciones en las clases implícitas

1. Las clases implícitas solo pueden existir dentro de otra clase, objeto o rasgo.
2. Las clases implícitas solo pueden tener un parámetro de constructor primario no implícito.
3. Puede que no haya otra definición de objeto, clase, rasgo o miembro de clase dentro del mismo ámbito que tenga el mismo nombre que la clase implícita.

Examples

Conversión implícita

Una conversión implícita permite al compilador convertir automáticamente un objeto de un tipo a otro tipo. Esto permite que el código trate un objeto como un objeto de otro tipo.

```
case class Foo(i: Int)

// without the implicit
Foo(40) + 2    // compilation-error (type mismatch)

// defines how to turn a Foo into an Int
implicit def fooToInt(foo: Foo): Int = foo.i

// now the Foo is converted to Int automatically when needed
Foo(40) + 2    // 42
```

La conversión es unidireccional: en este caso, no puede convertir `42` nuevo a `Foo(42)` . Para ello, se debe definir una segunda conversión implícita:

```
implicit def intToFoo(i: Int): Foo = Foo(i)
```

Tenga en cuenta que este es el mecanismo por el cual se puede agregar un valor flotante a un valor entero, por ejemplo.

> Las conversiones implícitas se deben usar con moderación porque ocultan lo que está sucediendo. Es una práctica recomendada utilizar una conversión explícita a través de una llamada de método, a menos que haya una ganancia de legibilidad tangible al usar una conversión implícita.

No hay un impacto significativo en el rendimiento de las conversiones implícitas.

Scala importa automáticamente una variedad de conversiones implícitas en `scala.Predef`, incluidas todas las conversiones de Java a Scala y viceversa. Estos están incluidos por defecto en cualquier compilación de archivos.

Parámetros implícitos

Los parámetros implícitos pueden ser útiles si un parámetro de un tipo debe definirse una vez en el alcance y luego aplicarse a todas las funciones que usan un valor de ese tipo.

Una llamada de función normal se parece a esto:

```
// import the duration methods
import scala.concurrent.duration._

// a normal method:
def doLongRunningTask(timeout: FiniteDuration): Long = timeout.toMillis

val timeout = 1.second
// timeout: scala.concurrent.duration.FiniteDuration = 1 second

// to call it
doLongRunningTask(timeout) // 1000
```

Ahora digamos que tenemos algunos métodos que tienen una duración de tiempo de espera, y queremos llamar a todos esos métodos utilizando el mismo tiempo de espera. Podemos definir el tiempo de espera como una variable implícita.

```
// import the duration methods
import scala.concurrent.duration._

// dummy methods that use the implicit parameter
def doLongRunningTaskA()(implicit timeout: FiniteDuration): Long = timeout.toMillis
def doLongRunningTaskB()(implicit timeout: FiniteDuration): Long = timeout.toMillis

// we define the value timeout as implicit
implicit val timeout: FiniteDuration = 1.second

// we can now call the functions without passing the timeout parameter
doLongRunningTaskA() // 1000
doLongRunningTaskB() // 1000
```

La forma en que funciona es que el compilador de scalac busca un valor en el ámbito que está **marcado como implícito** y **cuyo tipo coincide con** el del parámetro implícito. Si encuentra uno, lo aplicará como parámetro implícito.

> Tenga en cuenta que esto no funcionará si define dos o incluso más implicaciones del

mismo tipo en el ámbito.

Para personalizar el mensaje de error, utilice el `implicitNotFound` anotación del tipo:

```
@annotation.implicitNotFound(msg = "Select the proper implicit value for type M[${A}]!")
case class M[A](v: A) {}

def usage[O](implicit x: M[O]): O = x.v

//Does not work because no implicit value is present for type `M[Int]`
//usage[Int]   //Select the proper implicit value for type M[Int]!
implicit val first: M[Int] = M(1)
usage[Int]     //Works when `second` is not in scope
implicit val second: M[Int] = M(2)
//Does not work because more than one implicit values are present for the type `M[Int]`
//usage[Int]   //Select the proper implicit value for type M[Int]!
```

Un tiempo de espera es un caso de uso habitual para esto, o por ejemplo, en Akka, el sistema Actor es (la mayoría de las veces) siempre el mismo, por lo que generalmente se pasa implícitamente. Otro caso de uso sería diseño de la biblioteca, más comúnmente con las bibliotecas de PF que se basan en clases de tipos (como scalaz , gatos o éxtasis).

> Generalmente se considera una mala práctica usar parámetros implícitos con tipos básicos como *Int* , *Long* , *String* , etc., ya que creará confusión y hará que el código sea menos legible.

Clases Implícitas

Las clases implícitas permiten agregar nuevos métodos a las clases previamente definidas.

La clase `String` no tiene método sin `withoutVowels` . Esto se puede agregar así:

```
object StringUtil {
  implicit class StringEnhancer(str: String) {
    def withoutVowels: String = str.replaceAll("[aeiou]", "")
  }
}
```

La clase implícita tiene un único parámetro de constructor (`str`) con el tipo que le gustaría extender (`String`) y contiene el método que le gustaría "agregar" al tipo (`withoutVowels`). Los métodos recién definidos ahora se pueden usar directamente en el tipo mejorado (cuando el tipo mejorado está en el alcance implícito):

```
import StringUtil.StringEnhancer // Brings StringEnhancer into implicit scope

println("Hello world".withoutVowels) // Hll wrld
```

Bajo el capó, las clases implícitas definen una conversión implícita del tipo mejorado a la clase implícita, como esto:

```
implicit def toStringEnhancer(str: String): StringEnhancer = new StringEnhancer(str)
```

Las clases implícitas a menudo se definen como clases de valor para evitar la creación de objetos de tiempo de ejecución y, por lo tanto, eliminar la sobrecarga del tiempo de ejecución:

```
implicit class StringEnhancer(val str: String) extends AnyVal {
    /* conversions code here */
}
```

Con la definición mejorada anterior, no es necesario crear una nueva instancia de `StringEnhancer` cada vez que se invoca el método `withoutVowels` .

Resolución de parámetros implícitos usando 'implícitamente'

Suponiendo una lista de parámetros implícitos con más de un parámetro implícito:

```
case class Example(p1:String, p2:String)(implicit ctx1:SomeCtx1, ctx2:SomeCtx2)
```

Ahora, asumiendo que una de las instancias implícitas no está disponible (`SomeCtx1`) mientras que todas las demás instancias implícitas necesarias están dentro del alcance, para crear una instancia de la clase, se debe proporcionar una instancia de `SomeCtx1` .

Esto se puede hacer al mismo tiempo que se conserva una instancia implícita dentro del alcance utilizando la palabra clave `implicitly` :

```
Example("something","somethingElse")(new SomeCtx1(), implicitly[SomeCtx2])
```

Implicados en el REPL

Para ver todas las `implicits` en el alcance durante una sesión REPL:

```
scala> :implicits
```

Para incluir también las conversiones implícitas definidas en `Predef.scala` :

```
scala> :implicits -v
```

Si uno tiene una expresión y desea ver el efecto de todas las reglas de reescritura que se le aplican (incluidas las implícitas):

```
scala> reflect.runtime.universe.reify(expr) // No quotes. reify is a macro operating directly
on code.
```

(Ejemplo:

```
scala> import reflect.runtime.universe._
scala> reify(Array("Alice", "Bob", "Eve").mkString(", "))
resX: Expr[String] = Expr[String](Predef.refArrayOps(Array.apply("Alice", "Bob",
"Eve")(Predef.implicitly)).mkString(", "))
```

)

Lea Implícitos en línea: https://riptutorial.com/es/scala/topic/1732/implicitos

Capítulo 25: Inferencia de tipos

Examples

Inferencia de tipo local

Scala tiene un poderoso mecanismo de inferencia de tipos integrado al lenguaje. Este mecanismo se denomina 'Inferencia de tipo local':

```
val i = 1 + 2                 // the type of i is Int
val s = "I am a String"       // the type of s is String
def squared(x : Int) = x*x    // the return type of squared is Int
```

El compilador puede inferir el tipo de variables de la expresión de inicialización. De manera similar, el tipo de retorno de los métodos se puede omitir, ya que son equivalentes al tipo devuelto por el cuerpo del método. Los ejemplos anteriores son equivalentes a las declaraciones de tipo explícitas a continuación:

```
val i: Int = 1 + 2
val s: String = "I am a String"
def squared(x : Int): Int = x*x
```

Tipo de Inferencia y Genéricos

El compilador Scala también puede deducir parámetros de tipo cuando se llaman métodos polimórficos, o cuando se crean instancias de clases genéricas:

```
case class InferedPair[A, B](a: A, b: B)

val pairFirstInst = InferedPair("Husband", "Wife")  //type is InferedPair[String, String]

// Equivalent, with type explicitly defined
val pairSecondInst: InferedPair[String, String]
                      = InferedPair[String, String]("Husband", "Wife")
```

La forma anterior de inferencia de tipo es similar al Operador Diamond , introducido en Java 7.

Limitaciones a la inferencia

Hay escenarios en los que la inferencia de tipos de Scala no funciona. Por ejemplo, el compilador no puede inferir el tipo de parámetros del método:

```
def add(a, b) = a + b  // Does not compile
def add(a: Int, b: Int) = a + b // Compiles
def add(a: Int, b: Int): Int = a + b // Equivalent expression, compiles
```

El compilador no puede inferir el tipo de retorno de los métodos recursivos:

```
// Does not compile
def factorial(n: Int) = if (n == 0 || n == 1) 1 else n * factorial(n - 1)
// Compiles
def factorial(n: Int): Int = if (n == 0 || n == 1) 1 else n * factorial(n - 1)
```

La prevención de no inferir nada

Basado en esta entrada de blog .

Supongamos que tiene un método como este:

```
  def get[T]: Option[T] = ???
```

Cuando intenta llamarlo sin especificar el parámetro genérico, `Nothing` se deduce `Nothing` , lo que no es muy útil para una implementación real (y su resultado no es útil). Con la siguiente solución, el `NotNothing` contexto `NotNothing` puede evitar el uso del método sin especificar el tipo esperado (en este ejemplo, `RuntimeClass` también se excluye como para `ClassTags` no `Nothing` , pero `RuntimeClass` se infiere):

```
@implicitNotFound("Nothing was inferred")
sealed trait NotNothing[-T]

object NotNothing {
  implicit object notNothing extends NotNothing[Any]
  //We do not want Nothing to be inferred, so make an ambigous implicit
  implicit object `\n The error is because the type parameter was resolved to Nothing` extends
NotNothing[Nothing]
  //For classtags, RuntimeClass can also be inferred, so making that ambigous too
  implicit object `\n The error is because the type parameter was resolved to RuntimeClass`
extends NotNothing[RuntimeClass]
}

object ObjectStore {
  //Using context bounds
  def get[T: NotNothing]: Option[T] = {
    ???
  }

  def newArray[T](length: Int = 10)(implicit ct: ClassTag[T], evNotNothing: NotNothing[T]):
Option[Array[T]] = ???
}
```

Ejemplo de uso:

```
object X {
  //Fails to compile
  //val nothingInferred = ObjectStore.get

  val anOption = ObjectStore.get[String]
  val optionalArray = ObjectStore.newArray[AnyRef]()

  //Fails to compile
  //val runtimeClassInferred = ObjectStore.newArray()
}
```

Lea Inferencia de tipos en línea: https://riptutorial.com/es/scala/topic/4918/inferencia-de-tipos

Capítulo 26: Interoperabilidad de Java

Examples

Conversión de colecciones Scala a colecciones Java y viceversa

Cuando necesite pasar una colección a un método Java:

```
import scala.collection.JavaConverters._

val scalaList = List(1, 2, 3)
JavaLibrary.process(scalaList.asJava)
```

Si el código Java devuelve una colección Java, puede convertirlo en una colección Scala de una manera similar:

```
import scala.collection.JavaConverters._

val javaCollection = JavaLibrary.getList
val scalaCollection = javaCollection.asScala
```

Tenga en cuenta que estos son decoradores, por lo que simplemente envuelven las colecciones subyacentes en una interfaz de colección de Scala o Java. Por lo tanto, las llamadas `.asJava` y `.asScala` no copian las colecciones.

Arrays

Las matrices son matrices JVM regulares con un giro que se tratan como invariantes y tienen constructores especiales y conversiones implícitas. Constrúyelos sin la `new` palabra clave.

```
val a = Array("element")
```

Ahora tiene `a` tipo `Array[String]` .

```
val acs: Array[CharSequence] = a
//Error: type mismatch;  found   : Array[String]  required: Array[CharSequence]
```

Aunque `String` se puede convertir en `CharSequence` , `Array[String]` no se puede convertir en `Array[CharSequence]` .

Puede usar un `Array` como otras colecciones, gracias a una conversión implícita a `TraversableLike` `ArrayOps` :

```
val b: Array[Int] = a.map(_.length)
```

La mayoría de las colecciones de Scala (`TraversableOnce`) tienen un método `toArray` que toma un

`ClassTag` implícito para construir la matriz de resultados:

```
List(0).toArray
//> res1: Array[Int] = Array(0)
```

Esto facilita el uso de `TraversableOnce` en su código de Scala y luego lo pasa al código Java que espera una matriz.

Conversiones de tipo Scala y Java

Scala ofrece conversiones implícitas entre todos los principales tipos de colección en el objeto JavaConverters.

Las siguientes conversiones de tipo son bidireccionales.

Tipo Scala	Tipo de Java
Iterador	java.util.Iterador
Iterador	java.util.Enumeracion
Iterador	java.util.Iterable
Iterador	java.util.Coleccion
mutable.Buffer	java.util.List
mutable.Set	java.util.Set
mutable.Map	java.util.Map
mutable.ConcurrentMap	java.util.concurrent.ConcurrentMap

Algunas otras colecciones de Scala también se pueden convertir a Java, pero no tienen una conversión al tipo de Scala original:

Tipo Scala	Tipo de Java
Seq	java.util.List
mutable.Seq	java.util.List
Conjunto	java.util.Set
Mapa	java.util.Map

Referencia :

Conversiones entre colecciones de Java y Scala

Interfaces funcionales para funciones de Scala - scala-java8-compat

Un kit de compatibilidad de Java 8 para Scala.

La mayoría de los ejemplos se copian de Readme

Convertidores entre scala.FunctionN y java.util.function

```
import java.util.function._
import scala.compat.java8.FunctionConverters._

val foo: Int => Boolean = i => i > 7
def testBig(ip: IntPredicate) = ip.test(9)
println(testBig(foo.asJava))  // Prints true

val bar = new UnaryOperator[String]{ def apply(s: String) = s.reverse }
List("cod", "herring").map(bar.asScala)    // List("doc", "gnirrih")

def testA[A](p: Predicate[A])(a: A) = p.test(a)
println(testA(asJavaPredicate(foo))(4))  // Prints false
```

Convertidores entre las clases scala.Option y java.util Optional, OptionalDouble, OptionalInt y OptionalLong.

```
import scala.compat.java8.OptionConverters._

    class Test {
      val o = Option(2.7)
      val oj = o.asJava        // Optional[Double]
      val ojd = o.asPrimitive  // OptionalDouble
      val ojds = ojd.asScala   // Option(2.7) again
    }
```

Convertidores de colecciones de Scala a Java 8 Streams

```
import java.util.stream.IntStream

import scala.compat.java8.StreamConverters._
import scala.compat.java8.collectionImpl.{Accumulator, LongAccumulator}


  val m = collection.immutable.HashMap("fish" -> 2, "bird" -> 4)
  val parStream: IntStream = m.parValueStream
  val s: Int = parStream.sum
  // 6, potientially computed in parallel
  val t: List[String] = m.seqKeyStream.toScala[List]
  // List("fish", "bird")
  val a: Accumulator[(String, Int)] = m.accumulate // Accumulator[(String, Int)]

  val n = a.stepper.fold(0)(_ + _._1.length) +
    a.parStream.count // 8 + 2 = 10

  val b: LongAccumulator = java.util.Arrays.stream(Array(2L, 3L, 4L)).accumulate
  // LongAccumulator
  val l: List[Long] = b.to[List] // List(2L, 3L, 4L)
```

Lea Interoperabilidad de Java en línea:
https://riptutorial.com/es/scala/topic/2441/interoperabilidad-de-java

Capítulo 27: Interpolación de cuerdas

Observaciones

Esta característica existe en Scala 2.10.0 y superior.

Examples

Hola Interpolación De Cuerdas

El interpolador **s** permite el uso de variables dentro de una cadena.

```
val name = "Brian"
println(s"Hello $name")
```

Imprime "Hola Brian" en la consola cuando se ejecuta.

Interpolación de cadena formateada utilizando el Interpolador f

```
val num = 42d
```

Imprime dos decimales para `num` utilizando *f*

```
println(f"$num%2.2f")
42.00
```

Imprimir `num` usando notación científica usando *e*

```
println(f"$num%e")
4.200000e+01
```

Imprimir `num` en hexadecimal con *una*

```
println(f"$num%a")
0x1.5p5
```

Otras cadenas de formato se pueden encontrar en
https://docs.oracle.com/javase/6/docs/api/java/util/Formatter.html#detail

Usando la expresión en cadenas literales

Puede usar llaves para interpolar expresiones en literales de cadena:

```
def f(x: String) = x + x
val a = "A"
```

```
s"${a}"    // "A"
s"${f(a)}" // "AA"
```

Sin las llaves, Scala solo interpolaría el *identificador* después de `$` (en este caso `f`). Dado que no hay una conversión implícita de `f` en una `String` esta es una excepción en este ejemplo:

```
s"$f(a)"  // compile-time error (missing argument list for method f)
```

Interpoladores de cadena personalizados

Es posible definir interpoladores de cadena personalizados además de los integrados.

```
my"foo${bar}baz"
```

Es expandido por el compilador a:

```
new scala.StringContext("foo", "baz").my(bar)
```

`scala.StringContext` no tiene `my` método, por lo tanto, se puede proporcionar por conversión implícita. Un interpolador personalizado con el mismo comportamiento que la orden interna `s` interpolador entonces ser implementado de la siguiente manera:

```
implicit class MyInterpolator(sc: StringContext) {
  def my(subs: Any*): String = {
    val pit = sc.parts.iterator
    val sit = subs.iterator
    // Note parts.length == subs.length + 1
    val sb = new java.lang.StringBuilder(pit.next())
    while(sit.hasNext) {
      sb.append(sit.next().toString)
      sb.append(pit.next())
    }
    sb.toString
  }
}
```

Y la interpolación de `my"foo${bar}baz"` se desugaría a:

```
new MyInterpolation(new StringContext("foo", "baz")).my(bar)
```

Tenga en cuenta que no hay restricción en los argumentos o el tipo de retorno de la función de interpolación. Esto nos lleva por un camino oscuro donde la sintaxis de interpolación se puede usar creativamente para construir objetos arbitrarios, como se ilustra en el siguiente ejemplo:

```
case class Let(name: Char, value: Int)

implicit class LetInterpolator(sc: StringContext) {
  def let(value: Int): Let = Let(sc.parts(0).charAt(0), value)
}
```

```
let"a=${4}" // Let(a, 4)
let"b=${"foo"}" // error: type mismatch
let"c=" // error: not enough arguments for method let: (value: Int)Let
```

Interpoladores de cadenas como extractores.

También es posible utilizar la función de interpolación de cadenas de Scala para crear extractores elaborados (emparejadores de patrones), como quizás el más famoso empleado en la API de cuasiquotes de las macros de Scala.

Dado que `n"p0${i0}p1"` desaparece del `new StringContext("p0", "p1").n(i0)` , quizás no sea sorprendente que la funcionalidad del extractor esté habilitada al proporcionar una conversión implícita de `StringContext` a una clase con propiedad `n` de un tipo que define un método `unapply` o `unapplySeq` .

Como ejemplo, considere el siguiente extractor que extrae segmentos de ruta al construir una expresión regular a partir de las partes `StringContext` . Luego podemos delegar la mayor parte del trabajo pesado al método `unapplySeq` proporcionado por el resultado scala.util.matching.Regex :

```
implicit class PathExtractor(sc: StringContext) {
  object path {
    def unapplySeq(str: String): Option[Seq[String]] =
      sc.parts.map(Regex.quote).mkString("^", "([^/]+)", "$").r.unapplySeq(str)
  }
}

"/documentation/scala/1629/string-interpolation" match {
  case path"/documentation/${topic}/${id}/${_}" => println(s"$topic, $id")
  case _ => ???
}
```

Tenga en cuenta que el objeto de `path` también podría definir un método de `apply` para comportarse como un interpolador regular también.

Interpolación de cuerdas sin procesar

Puede utilizar el interpolador en **bruto** si desea que una cadena se imprima tal como está y sin ningún escape de literales.

```
println(raw"Hello World In English And French\nEnglish:\tHello World\nFrench:\t\tBonjour Le
Monde")
```

Con el uso del interpolador en **bruto** , debería ver lo siguiente impreso en la consola:

```
Hello World In English And French\nEnglish:\tHello World\nFrench:\t\tBonjour Le Monde
```

Sin el interpolador en **bruto** , `\n` `\t` se habrían escapado.

```
println("Hello World In English And French\nEnglish:\tHello World\nFrench:\t\tBonjour Le
Monde")
```

Huellas dactilares:

```
Hello World In English And French
English:       Hello World
French:        Bonjour Le Monde
```

Lea Interpolación de cuerdas en línea: https://riptutorial.com/es/scala/topic/1629/interpolacion-de-cuerdas

Capítulo 28: Invocación Dinámica

Introducción

Scala le permite usar la invocación dinámica al llamar a métodos o acceder a campos en un objeto. En lugar de tener esto integrado en el lenguaje, esto se logra a través de reglas de reescritura similares a las de las conversiones implícitas, habilitadas por el rasgo del marcador [`scala.Dynamic`] [Dynamic scaladoc]. Esto le permite emular la capacidad de agregar propiedades dinámicamente a objetos presentes en idiomas dinámicos, y más. [Scaladoc dinámico]: http://www.scala-lang.org/api/2.12.x/scala/Dynamic.html

Sintaxis

- clase Foo extiende Dynamic
- foo.field
- foo.field = valor
- foo.method (args)
- foo.method (namedArg = x, y)

Observaciones

Para declarar subtipos de `Dynamic` , la `dynamics` característica del lenguaje debe estar habilitada, ya sea importando `scala.language.dynamics` o mediante la opción del compilador `-language:dynamics` . Los usuarios de esta `Dynamic` que no definen sus propios subtipos no necesitan habilitar esto.

Examples

Accesos de campo

Esta:

```
class Foo extends Dynamic {
  // Expressions are only rewritten to use Dynamic if they are not already valid
  // Therefore foo.realField will not use select/updateDynamic
  var realField: Int = 5
  // Called for expressions of the type foo.field
  def selectDynamic(fieldName: String) = ???
  def updateDynamic(fieldName: String)(value: Int) = ???
}
```

permite un acceso simple a los campos:

```
val foo: Foo = ???
foo.realField // Does NOT use Dynamic; accesses the actual field
foo.realField = 10 // Actual field access here too
foo.unrealField // Becomes foo.selectDynamic(unrealField)
```

```
foo.field = 10  // Becomes foo.updateDynamic("field")(10)
foo.field = "10" // Does not compile; "10" is not an Int.
foo.x() // Does not compile; Foo does not define applyDynamic, which is used for methods.
foo.x.apply() // DOES compile, as Nothing is a subtype of () => Any
// Remember, the compiler is still doing static type checks, it just has one more way to
// "recover" and rewrite otherwise invalid code now.
```

Método de llamadas

Esta:

```
class Villain(val minions: Map[String, Minion]) extends Dynamic {
  def applyDynamic(name: String)(jobs: Task*) = jobs.foreach(minions(name).do)
  def applyDynamicNamed(name: String)(jobs: (String, Task)*) = jobs.foreach {
    // If a parameter does not have a name, and is simply given, the name passed as ""
    case ("", task) => minions(name).do(task)
    case (subsys, task) => minions(name).subsystems(subsys).do(task)
  }
}
```

Permite llamadas a métodos, con y sin parámetros nombrados:

```
val gru: Villain = ???
gru.blu() // Becomes gru.applyDynamic("blu")()
// Becomes gru.applyDynamicNamed("stu")(("fooer", ???), ("boomer", ???), ("", ???),
//          ("computer breaker", ???), ("fooer", ???))
// Note how the `???` without a name is given the name ""
// Note how both occurrences of `fooer` are passed to the method
gru.stu(fooer = ???, boomer = ???, ???, `computer breaker` = ???, fooer = ???)
gru.ERR("a") // Somehow, scalac thinks "a" is not a Task, though it clearly is (it isn't)
```

Interacción entre el acceso de campo y el método de actualización

Un poco contrario a la intuición (pero también la única forma sensata de hacerlo funcionar), esto:

```
val dyn: Dynamic = ???
dyn.x(y) = z
```

es equivalente a:

```
dyn.selectDynamic("x").update(y, z)
```

mientras

```
dyn.x(y)
```

es todavía

```
dyn.applyDynamic("x")(y)
```

Es importante ser consciente de esto, o de lo contrario puede pasar inadvertido y causar errores

extraños.

Lea Invocación Dinámica en línea: https://riptutorial.com/es/scala/topic/8296/invocacion-dinamica

Capítulo 29: Inyección de dependencia

Examples

Patrón de pastel con clase de implementación interna.

```
//create a component that will be injected
trait TimeUtil {
    lazy val timeUtil = new TimeUtilImpl()

    class TimeUtilImpl{
        def now() = new DateTime()
    }
}

//main controller is depended on time util
trait MainController {
    _ : TimeUtil => //inject time util into main controller

    lazy val mainController = new MainControllerImpl()

    class MainControllerImpl {
        def printCurrentTime() = println(timeUtil.now()) //timeUtil is injected from TimeUtil
trait
    }
}

object MainApp extends App {
    object app extends MainController
       with TimeUtil //wire all components

    app.mainController.printCurrentTime()
}
```

En el ejemplo anterior, demostré cómo inyectar TimeUtil en MainController.

La sintaxis más importante es la auto-anotación (`_: TimeUtil =>`) que consiste en inyectar `TimeUtil` en `MainController` . En otra palabra, `MainController` depende de `TimeUtil` .

Utilizo la clase interna (por ejemplo, `TimeUtilImpl`) en cada componente porque, en mi opinión, es más fácil de probar ya que podemos burlarnos de la clase interna. Y también es más fácil para rastrear desde donde se llama el método cuando el proyecto se vuelve más complejo.

Por último, conecto todos los componentes juntos. Si está familiarizado con Guice, esto es equivalente a `Binding`

Lea Inyección de dependencia en línea: https://riptutorial.com/es/scala/topic/5909/inyeccion-de-dependencia

Capítulo 30: JSON

Examples

JSON con spray-json

spray-json proporciona una manera fácil de trabajar con JSON. Usando formatos implícitos, todo sucede "detrás de escena":

Haga que la biblioteca esté disponible con SBT

Para administrar `spray-json` con las dependencias de la biblioteca administrada SBT :

```
libraryDependencies += "io.spray" %% "spray-json" % "1.3.2"
```

Tenga en cuenta que el último parámetro, el número de versión (`1.3.2`), puede ser diferente en diferentes proyectos.

La biblioteca `spray-json` está alojada en repo.spray.io .

Importar la biblioteca

```
import spray.json._
import DefaultJsonProtocol._
```

El protocolo JSON predeterminado `DefaultJsonProtocol` contiene formatos para todos los tipos básicos. Para proporcionar la funcionalidad JSON para tipos personalizados, use los constructores de conveniencia para los formatos o los formatos de escritura explícitamente.

Leer json

```
// generates an intermediate JSON representation (abstract syntax tree)
val res = """{ "foo": "bar" }""".parseJson // JsValue = {"foo":"bar"}

res.convertTo[Map[String, String]] // Map(foo -> bar)
```

Escribir json

```
val values = List("a", "b", "c")
```

```
values.toJson.prettyPrint // ["a", "b", "c"]
```

DSL

DSL no es compatible.

Lectura-escritura a clases de casos

El siguiente ejemplo muestra cómo serializar un objeto de clase de caso en el formato JSON.

```
case class Address(street: String, city: String)
case class Person(name: String, address: Address)

// create the formats and provide them implicitly
implicit val addressFormat = jsonFormat2(Address)
implicit val personFormat = jsonFormat2(Person)

// serialize a Person
Person("Fred", Address("Awesome Street 9", "SuperCity"))
val fredJsonString = fred.toJson.prettyPrint
```

Esto resulta en el siguiente JSON:

```
{
  "name": "Fred",
  "address": {
    "street": "Awesome Street 9",
    "city": "SuperCity"
  }
}
```

Ese JSON puede, a su vez, deserializarse de nuevo en un objeto:

```
val personRead = fredJsonString.parseJson.convertTo[Person]
//Person(Fred,Address(Awesome Street 9,SuperCity))
```

Formato personalizado

Escriba un `JsonFormat` personalizado si se `JsonFormat` una serialización especial de un tipo. Por ejemplo, si los nombres de campo son diferentes en Scala que en JSON. O, si diferentes tipos de concreto son instanciados basados en la entrada.

```
implicit object BetterPersonFormat extends JsonFormat[Person] {
    // deserialization code
    override def read(json: JsValue): Person = {
        val fields = json.asJsObject("Person object expected").fields
        Person(
            name = fields("name").convertTo[String],
```

```
            address = fields("home").convertTo[Address]
        )
    }

    // serialization code
    override def write(person: Person): JsValue = JsObject(
        "name" -> person.name.toJson,
        "home" -> person.address.toJson
    )
}
```

JSON con Circe

Circe proporciona códecs derivados de tiempo de compilación para en / decode json en clases de casos. Un ejemplo simple se ve así:

```
import io.circe._
import io.circe.generic.auto._
import io.circe.parser._
import io.circe.syntax._

case class User(id: Long, name: String)

val user = User(1, "John Doe")

// {"id":1,"name":"John Doe"}
val json = user.asJson.noSpaces

// Right(User(1L, "John Doe"))
val res: Either[Error, User] = decode[User](json)
```

JSON con play-json

play-json usa formatos implícitos como otros frameworks json

Dependencia de SBT: `libraryDependencies += ""com.typesafe.play" %% "play-json" % "2.4.8"`

```
import play.api.libs.json._
import play.api.libs.functional.syntax._ // if you need DSL
```

`DefaultFormat` contiene formatos predeterminados para leer / escribir todos los tipos básicos. Para proporcionar la funcionalidad JSON para sus propios tipos, puede usar constructores de conveniencia para formatos o escribir formatos explícitamente.

Leer json

```
// generates an intermediate JSON representation (abstract syntax tree)
val res = Json.parse("""{ "foo": "bar" }""") // JsValue = {"foo":"bar"}

res.as[Map[String, String]]                  // Map(foo -> bar)
res.validate[Map[String, String]]            //JsSuccess(Map(foo -> bar),)
```

Escribir json

```
val values = List("a", "b", "c")
Json.stringify(Json.toJson(values))          // ["a", "b", "c"]
```

DSL

```
val json = parse("""{ "foo": [{"foo": "bar"}]}""")
(json \ "foo").get                    //Simple path: [{"foo":"bar"}]
(json \\ "foo")                       //Recursive path:List([{"foo":"bar"}], "bar")
(json \ "foo")(0).get                  //Index lookup (for JsArrays): {"foo":"bar"}
```

Como siempre, prefiera la coincidencia de patrones con `JsSuccess` / `JsError` e intente evitar las `.get` , `array(i)` .

Leer y escribir a clase de caso

```
case class Address(street: String, city: String)
case class Person(name: String, address: Address)

// create the formats and provide them implicitly
implicit val addressFormat = Json.format[Address]
implicit val personFormat = Json.format[Person]

// serialize a Person
val fred = Person("Fred", Address("Awesome Street 9", "SuperCity"))
val fredJsonString = Json.stringify(Json.toJson(Json.toJson(fred)))

val personRead = Json.parse(fredJsonString).as[Person] //Person(Fred,Address(Awesome Street
9,SuperCity))
```

Formato propio

Puede escribir su propio JsonFormat si necesita una serialización especial de su tipo (por ejemplo, nombre los campos de forma diferente en scala y Json o ejemplifique diferentes tipos concretos en función de la entrada)

```
case class Address(street: String, city: String)

// create the formats and provide them implicitly
implicit object AddressFormatCustom extends Format[Address] {
  def reads(json: JsValue): JsResult[Address] = for {
    street <- (json \ "Street").validate[String]
    city <- (json \ "City").validate[String]
  } yield Address(street, city)

  def writes(x: Address): JsValue = Json.obj(
    "Street" -> x.street,
    "City" -> x.city
  )
}
// serialize an address
val address = Address("Awesome Street 9", "SuperCity")
val addressJsonString = Json.stringify(Json.toJson(Json.toJson(address)))
//{"Street":"Awesome Street 9","City":"SuperCity"}

val addressRead = Json.parse(addressJsonString).as[Address]
//Address(Awesome Street 9,SuperCity)
```

Alternativa

Si el json no coincide exactamente con los campos de su clase de caso (`isAlive` en la clase de caso vs `is_alive` en json):

```
case class User(username: String, friends: Int, enemies: Int, isAlive: Boolean)

object User {

  import play.api.libs.functional.syntax._
  import play.api.libs.json._

  implicit val userReads: Reads[User] = (
      (JsPath \ "username").read[String] and
      (JsPath \ "friends").read[Int] and
      (JsPath \ "enemies").read[Int] and
      (JsPath \ "is_alive").read[Boolean]
    ) (User.apply _)
}
```

Json con campos opcionales

```
case class User(username: String, friends: Int, enemies: Int, isAlive: Option[Boolean])

object User {

  import play.api.libs.functional.syntax._
  import play.api.libs.json._

  implicit val userReads: Reads[User] = (
      (JsPath \ "username").read[String] and
      (JsPath \ "friends").read[Int] and
      (JsPath \ "enemies").read[Int] and
      (JsPath \ "is_alive").readNullable[Boolean]
    ) (User.apply _)
}
```

Lectura de marcas de tiempo de json

Imagina que tienes un objeto Json, con un campo de marca de tiempo Unix:

```
{
  "field": "example field",
  "date": 1459014762000
}
```

solución:

```
case class JsonExampleV1(field: String, date: DateTime)
object JsonExampleV1{
  implicit val r: Reads[JsonExampleV1] = (
    (__ \ "field").read[String] and
      (__ \ "date").read[DateTime](Reads.DefaultJodaDateReads)
    )(JsonExampleV1.apply _)
}
```

Leer clases de casos personalizados

Ahora, si envuelve sus identificadores de objetos para la seguridad de tipos, disfrutará esto. Vea el siguiente objeto json:

```
{
  "id": 91,
  "data": "Some data"
}
```

y las clases de casos correspondientes:

```
case class MyIdentifier(id: Long)

case class JsonExampleV2(id: MyIdentifier, data: String)
```

Ahora solo necesita leer el tipo primitivo (Largo) y asignarlo a su idenfier:

```
object JsonExampleV2 {
  implicit val r: Reads[JsonExampleV2] = (
      (__ \ "id").read[Long].map(MyIdentifier) and
    (__ \ "data").read[String]
    )(JsonExampleV2.apply _)
}
```

código en https://github.com/pedrorijo91/scala-play-json-examples

JSON con json4s

json4s usa formatos implícitos como otros frameworks json.

Dependencia de SBT:

```
libraryDependencies += "org.json4s" %% "json4s-native" % "3.4.0"
//or
libraryDependencies += "org.json4s" %% "json4s-jackson" % "3.4.0"
```

Importaciones

```
import org.json4s.JsonDSL._
import org.json4s._
import org.json4s.native.JsonMethods._

implicit val formats = DefaultFormats
```

`DefaultFormats` contiene formatos predeterminados para leer / escribir todos los tipos básicos.

Leer json

```
// generates an intermediate JSON representation (abstract syntax tree)
val res = parse("""{ "foo": "bar" }""")           // JValue = {"foo":"bar"}
```

```
res.extract[Map[String, String]]                  // Map(foo -> bar)
```

Escribir json

```
val values = List("a", "b", "c")
compact(render(values))          // ["a", "b", "c"]
```

DSL

```
json \ "foo"       //Simple path: JArray(List(JObject(List((foo,JString(bar))))))
json \\ "foo"      //Recursive path: ~List([{"foo":"bar"}], "bar")
(json \ "foo")(0)  //Index lookup (for JsArrays): JObject(List((foo,JString(bar))))
("foo" -> "bar") ~ ("field" -> "value")    // {"foo":"bar","field":"value"}
```

Leer y escribir a clase de caso

```
import org.json4s.native.Serialization.{read, write}

case class Address(street: String, city: String)
val addressString = write(Address("Awesome stree", "Super city"))
// {"street":"Awesome stree","city":"Super city"}

read[Address](addressString) // Address(Awesome stree,Super city)
//or
parse(addressString).extract[Address]
```

Leer y escribir listas heterogéneas

Para serializar y deserializar una lista heterogénea (o polimórfica), se deben proporcionar sugerencias de tipo específicas.

```
trait Location
case class Street(name: String) extends Location
case class City(name: String, zipcode: String) extends Location
case class Address(street: Street, city: City) extends Location
case class Locations (locations : List[Location])

implicit val formats = Serialization.formats(ShortTypeHints(List(classOf[Street],
classOf[City], classOf[Address])))

val locationsString = write(Locations(Street("Lavelle Street"):: City("Super city","74658")))

read[Locations](locationsString)
```

Formato propio

```
class AddressSerializer extends CustomSerializer[Address](format => (
  {
    case JObject(JField("Street", JString(s)) :: JField("City", JString(c)) :: Nil) =>
      new Address(s, c)
  },
  {
    case x: Address => ("Street" -> x.street) ~ ("City" -> x.city)
  }
```

```
  ))

implicit val formats = DefaultFormats + new AddressSerializer
val str = write[Address](Address("Awesome Stree", "Super City"))
// {"Street":"Awesome Stree","City":"Super City"}
read[Address](str)
// Address(Awesome Stree,Super City)
```

Lea JSON en línea: https://riptutorial.com/es/scala/topic/2348/json

Capítulo 31: La coincidencia de patrones

Sintaxis

- selector match partialFunction
- selector de coincidencia {lista de alternativas de casos) // Esta es la forma más común de las anteriores

Parámetros

Parámetro	Detalles
selector	La expresión cuyo valor se corresponde con el patrón.
alternativas	una lista de alternativas delimitadas por `case` .

Examples

Coincidencia de patrón simple

Este ejemplo muestra cómo hacer coincidir una entrada con varios valores:

```
def f(x: Int): String = x match {
  case 1 => "One"
  case 2 => "Two"
  case _ => "Unknown!"
}

f(2)  // "Two"
f(3)  // "Unknown!"
```

Demo en vivo

Nota: _ es la *caída a través* o *por defecto* caso, pero no es necesario.

```
def g(x: Int): String = x match {
  case 1 => "One"
  case 2 => "Two"
}

g(1)  // "One"
g(3)  // throws a MatchError
```

Para evitar lanzar una excepción, es una mejor práctica de programación funcional aquí manejar el caso predeterminado (`case _ => <do something>`). Tenga en cuenta que hacer coincidir *una clase de caso* puede ayudar al compilador a producir una advertencia si falta un caso. Lo mismo ocurre con los tipos definidos por el usuario que extienden un rasgo sellado. Si la coincidencia es

total, puede que no sea necesario un caso predeterminado

También es posible hacer coincidencias con valores que no están definidos en línea. Estos deben ser *identificadores estables* , que se obtienen utilizando un nombre en mayúscula o encerrando backticks.

Con `One` y `two` definidos en otro lugar, o pasados como parámetros de función:

```
val One: Int = 1
val two: Int = 2
```

Se pueden emparejar contra de la siguiente manera:

```
def g(x: Int): String = x match {
  case One => "One"
  case `two` => "Two"
}
```

A diferencia de otros lenguajes de programación como Java, por ejemplo, no hay caída. Si un bloque de caso coincide con una entrada, se ejecuta y se completa la coincidencia. Por lo tanto, el caso menos específico debería ser el último bloque de caso.

```
def f(x: Int): String = x match {
  case _ => "Default"
  case 1 => "One"
}

f(5) // "Default"
f(1) // "Default"
```

Coincidencia de patrones con identificador estable

En la coincidencia de patrón estándar, el identificador utilizado sombreará cualquier identificador en el ámbito de envolvente. A veces es necesario hacer coincidir en la variable del ámbito de cierre.

La siguiente función de ejemplo toma un carácter y una lista de tuplas y devuelve una nueva lista de tuplas. Si el carácter existía como el primer elemento en una de las tuplas, el segundo elemento se incrementa. Si aún no existe en la lista, se crea una nueva tupla.

```
def tabulate(char: Char, tab: List[(Char, Int)]): List[(Char, Int)] = tab match {
  case Nil => List((char, 1))
  case (`char`, count) :: tail => (char, count + 1) :: tail
  case head :: tail => head :: tabulate(char, tail)
}
```

Lo anterior demuestra la coincidencia de patrones donde la entrada del método, `char` , se mantiene 'estable' en la coincidencia de patrones: es decir, si llama a `tabulate('x', ...)` , la primera declaración de caso se interpretaría como:

```
case('x', count) => ...
```

Scala interpretará cualquier variable demarcada con una marca de verificación como un identificador estable: también interpretará cualquier variable que comience con una letra mayúscula de la misma manera.

Patrón de coincidencia en una secuencia

Para verificar un número preciso de elementos en la colección

```
def f(ints: Seq[Int]): String = ints match {
  case Seq() =>
      "The Seq is empty !"
  case Seq(first) =>
      s"The seq has exactly one element : $first"
  case Seq(first, second) =>
      s"The seq has exactly two elements : $first, $second"
  case  s @ Seq(_, _, _) =>
      s"s is a Seq of length three and looks like ${s}"  // Note individual elements are not
bound to their own names.
  case s: Seq[Int] if s.length == 4 =>
      s"s is a Seq of Ints of exactly length 4"  // Again, individual elements are not bound
to their own names.
  case _ =>
      "No match was found!"
}
```

Demo en vivo

Para extraer el (los) primer (s) elemento (s) y guardar el resto como una colección:

```
def f(ints: Seq[Int]): String = ints match {
  case Seq(first, second, tail @ _*) =>
      s"The seq has at least two elements : $first, $second. The rest of the Seq is $tail"
  case Seq(first, tail @ _*) =>
      s"The seq has at least one element : $first. The rest of the Seq is $tail"
  // alternative syntax
  // here of course this one will never match since it checks
  // for the same thing as the one above
  case first +: tail =>
      s"The seq has at least one element : $first. The rest of the Seq is $tail"
  case _ =>
      "The seq didn't match any of the above, so it must be empty"
}
```

En general, cualquier forma que se pueda usar para construir una secuencia se puede usar para hacer una comparación de patrones con una secuencia existente.

Tenga en cuenta que mientras usa `Nil` y `::` funcionará cuando el patrón coincida con una secuencia, la convierte a una `List` y puede tener resultados inesperados. Limítate a `Seq( ...)` y `+:` para evitar esto.

Tenga en cuenta que mientras usa `::` no funcionará para `WrappedArray` , `Vector` , etc., vea:

```
scala> def f(ints:Seq[Int]) = ints match {
     | case h :: t => h
     | case _ => "No match"
     | }
f: (ints: Seq[Int])Any

scala> f(Array(1,2))
res0: Any = No match
```

Y con `+:`:

```
scala> def g(ints:Seq[Int]) = ints match {
     | case h+:t => h
     | case _ => "No match"
     | }
g: (ints: Seq[Int])Any

scala> g(Array(1,2).toSeq)
res4: Any = 1
```

Guardias (si son expresiones)

Las declaraciones de casos se pueden combinar con las expresiones if para proporcionar lógica adicional cuando coinciden los patrones.

```
def checkSign(x: Int): String = {
    x match {
      case a if a < 0 => s"$a is a negative number"
      case b if b > 0 => s"$b is a positive number"
      case c => s"$c neither positive nor negative"
    }
}
```

Es importante asegurarse de que sus guardias no creen una coincidencia no exhaustiva (el compilador a menudo no lo detectará):

```
def f(x: Option[Int]) = x match {
    case Some(i) if i % 2 == 0 => doSomething(i)
    case None    => doSomethingIfNone
}
```

Esto lanza un `MatchError` en los números impares. Debe tener en cuenta todos los casos, o usar un caso de coincidencia de comodín:

```
def f(x: Option[Int]) = x match {
    case Some(i) if i % 2 == 0 => doSomething(i)
    case _ => doSomethingIfNoneOrOdd
}
```

Coincidencia de patrones con clases de casos

Cada clase de caso define un extractor que puede usarse para capturar a los miembros de la

clase de caso cuando coincida el patrón:

```
case class Student(name: String, email: String)

def matchStudent1(student: Student): String = student match {
    case Student(name, email) => s"$name has the following email: $email" // extract name and
email
}
```

Se aplican todas las reglas normales de coincidencia de patrones: puede usar protecciones y expresiones constantes para controlar la coincidencia:

```
def matchStudent2(student: Student): String = student match {
    case Student("Paul", _) => "Matched Paul" // Only match students named Paul, ignore email
    case Student(name, _) if name == "Paul" => "Matched Paul" // Use a guard to match students
named Paul, ignore email
    case s if s.name == "Paul" => "Matched Paul" // Don't use extractor; use a guard to match
students named Paul, ignore email
    case Student("Joe", email) => s"Joe has email $email" // Match students named Joe, capture
their email
    case Student(name, email) if name == "Joe" => s"Joe has email $email" // use a guard to
match students named Joe, capture their email
    case Student(name, email) => s"$name has email $email." // Match all students, capture
name and email
}
```

Coincidencia en una opción

Si está emparejando en un tipo de opción :

```
def f(x: Option[Int]) = x match {
    case Some(i) => doSomething(i)
    case None    => doSomethingIfNone
}
```

Esto es funcionalmente equivalente a usar `fold` , o `map` / `getOrElse` :

```
def g(x: Option[Int]) = x.fold(doSomethingIfNone)(doSomething)
def h(x: Option[Int]) = x.map(doSomething).getOrElse(doSomethingIfNone)
```

Rasgos sellados a juego del patrón

Cuando el patrón coincida con un objeto cuyo tipo es un rasgo sellado, Scala verificará en el momento de la compilación que todos los casos están "completamente emparejados":

```
sealed trait Shape
case class Square(height: Int, width: Int) extends Shape
case class Circle(radius: Int) extends Shape
case object Point extends Shape


def matchShape(shape: Shape): String = shape match {
    case Square(height, width) => "It's a square"
```

```
    case Circle(radius)        => "It's a circle"
    //no case for Point because it would cause a compiler warning.
}
```

Si posteriormente se agrega una nueva `case class` para `Shape` , todas las declaraciones de `match` en `Shape` comenzarán a mostrar una advertencia del compilador. Esto facilita la refactorización completa: el compilador alertará al desarrollador de todo el código que debe actualizarse.

Coincidencia de patrones con Regex

```
val emailRegex: Regex = "(.+)@(.+)\\.(.+)".r

"name@example.com" match {
  case emailRegex(userName, domain, topDomain) => println(s"Hi $userName from $domain")
  case _ => println(s"This is not a valid email.")
}
```

En este ejemplo, la expresión regular intenta coincidir con la dirección de correo electrónico proporcionada. Si lo hace, se extrae e imprime el nombre de `userName` y el `domain` . `topDomain` también se extrae, pero nada se hace con él en este ejemplo. Llamando `.r` en una cadena `str` es equivalente a la `new Regex(str)` . La función `r` está disponible a través de una conversión implícita .

Carpeta de patrones (@)

El signo `@` vincula una variable a un nombre durante una coincidencia de patrón. La variable enlazada puede ser todo el objeto coincidente o parte del objeto coincidente:

```
sealed trait Shape
case class Rectangle(height: Int, width: Int) extends Shape
case class Circle(radius: Int) extends Shape
case object Point extends Shape

(Circle(5): Shape) match {
  case Rectangle(h, w) => s"rectangle, $h x $w."
  case Circle(r) if r > 9 => s"large circle"
  case c @ Circle(_) => s"small circle: ${c.radius}"  // Whole matched object is bound to c
  case Point => "point"
}
```

```
> res0: String = small circle: 5
```

El identificador enlazado se puede utilizar en filtros condicionales. Así:

```
case Circle(r) if r > 9 => s"large circle"
```

Se puede escribir como:

```
case c @ Circle(_) if c.radius > 9 => s"large circle"
```

El nombre se puede vincular solo a una parte del patrón coincidente:

```
Seq(Some(1), Some(2), None) match {
  // Only the first element of the matched sequence is bound to the name 'c'
  case Seq(c @ Some(1), _*) => head
  case _ => None
}
```

```
> res0: Option[Int] = Some(1)
```

Tipos de coincidencia de patrones

La coincidencia de patrones también se puede usar para verificar el tipo de una instancia, en lugar de usar `isInstanceOf[B]` :

```
val anyRef: AnyRef = ""

anyRef match {
  case _: Number       => "It is a number"
  case _: String       => "It is a string"
  case _: CharSequence => "It is a char sequence"
}
//> res0: String = It is a string
```

El orden de los casos es importante:

```
anyRef match {
  case _: Number       => "It is a number"
  case _: CharSequence => "It is a char sequence"
  case _: String       => "It is a string"
}
//> res1: String = It is a char sequence
```

De esta manera, es similar a una declaración clásica de "cambio", sin la funcionalidad de acceso directo. Sin embargo, también puede establecer patrones de coincidencia y valores de 'extracción' del tipo en cuestión. Por ejemplo:

```
case class Foo(s: String)
case class Bar(s: String)
case class Woo(s: String, i: Int)

def matcher(g: Any):String = {
  g match {
    case Bar(s) => s + " is classy!"
    case Foo(_) => "Someone is wicked smart!"
    case Woo(s, _) => s + " is adventerous!"
    case _ => "What are we talking about?"
  }
}

print(matcher(Foo("Diana")))  // prints 'Diana is classy!'
print(matcher(Bar("Hadas")))  // prints 'Someone is wicked smart!'
print(matcher(Woo("Beth", 27)))   // prints 'Beth is adventerous!'
print(matcher(Option("Katie")))  // prints 'What are we talking about?'
```

Tenga en cuenta que en el caso de `Foo` y `Woo` usamos el guión bajo (`_`) para "coincidir con una variable no vinculada". Es decir, el valor (en este caso `Hadas` y `27` , respectivamente) no está

vinculado a un nombre y, por lo tanto, no está disponible en el controlador para ese caso. Esta es una taquigrafía útil para hacer coincidir el valor 'cualquiera' sin preocuparse por cuál es ese valor.

Coincidencia de patrones compilada como conmutador de tablas o de búsqueda

La anotación `@switch` le dice al compilador que la declaración de `match` se puede reemplazar con una sola instrucción de `tableswitch` en el nivel de bytecode. Esta es una optimización menor que puede eliminar comparaciones innecesarias y cargas variables durante el tiempo de ejecución.

La anotación `@switch` funciona solo para coincidencias contra constantes literales e identificadores de `final val` . Si la coincidencia de patrón no se puede compilar como un `tableswitch` / `lookupswitch` , el compilador mostrará una advertencia.

```
import annotation.switch

def suffix(i: Int) = (i: @switch) match {
  case 1 => "st"
  case 2 => "nd"
  case 3 => "rd"
  case _ => "th"
}
```

Los resultados son los mismos que en una coincidencia de patrón normal:

```
scala> suffix(2)
res1: String = "2nd"

scala> suffix(4)
res2: String = "4th"
```

De la Documentación Scala (2.8+) - `@switch` :

> Una anotación que se aplicará a una expresión de coincidencia. Si está presente, el compilador verificará que la coincidencia se haya compilado en un conmutador de tablas o un interruptor de búsqueda, y emitirá un error si en su lugar se compila en una serie de expresiones condicionales.

De la especificación de Java:

- tableswitch : "Accede a la tabla de saltos por índice y salta"
- interruptor de búsqueda : "Acceda a la tabla de saltos por coincidencia de teclas y salte"

Combinando múltiples patrones a la vez

El `|` se puede usar para que una única sentencia de caso coincida con varias entradas para obtener el mismo resultado:

```
def f(str: String): String = str match {
```

```
  case "foo" | "bar" => "Matched!"
  case _ => "No match."
}

f("foo")  // res0: String = Matched!
f("bar")  // res1: String = Matched!
f("fubar")  // res2: String = No match.
```

Tenga en cuenta que si bien los **valores** coincidentes funcionan de esta manera, la siguiente coincidencia de **tipos** causará problemas:

```
sealed class FooBar
case class Foo(s: String) extends FooBar
case class Bar(s: String) extends FooBar

val d = Foo("Diana")
val h = Bar("Hadas")

// This matcher WILL NOT work.
def matcher(g: FooBar):String = {
  g match {
    case Foo(s) | Bar(s) => print(s)  // Won't work: s cannot be resolved
    case Foo(_) | Bar(_) => _         // Won't work: _ is an unbound placeholder
    case _ => "Could not match"
  }
}
```

Si en el último caso (con _) no necesita el valor de la variable no vinculada y solo quiere hacer otra cosa, está bien:

```
def matcher(g: FooBar):String = {
  g match {
    case Foo(_) | Bar(_) => "Is either Foo or Bar."  // Works fine
    case _ => "Could not match"
  }
}
```

De lo contrario, te quedas con la división de sus casos:

```
def matcher(g: FooBar):String = {
  g match {
    case Foo(s) => s
    case Bar(s) => s
    case _ => "Could not match"
  }
}
```

Coincidencia de patrones en tuplas

Dada la siguiente `List` de tuplas:

```
val pastries = List(("Chocolate Cupcake", 2.50),
                    ("Vanilla Cupcake", 2.25),
                    ("Plain Muffin", 3.25))
```

La comparación de patrones se puede utilizar para manejar cada elemento de manera diferente:

```
pastries foreach { pastry =>
  pastry match {
    case ("Plain Muffin", price) => println(s"Buying muffin for $price")
    case p if p._1 contains "Cupcake" => println(s"Buying cupcake for ${p._2}")
    case _ => println("We don't sell that pastry")
  }
}
```

El primer caso muestra cómo hacer coincidir una cadena específica y obtener el precio correspondiente. El segundo caso muestra el uso de if y la extracción de la tupla para que coincida con los elementos de la tupla.

Lea La coincidencia de patrones en línea: https://riptutorial.com/es/scala/topic/661/la-coincidencia-de-patrones

Capítulo 32: Macros

Introducción

Las macros son una forma de metaprogramación en tiempo de compilación. Ciertos elementos del código Scala, como las anotaciones y los métodos, pueden transformarse en otro código cuando se compilan. Las macros son códigos Scala normales que operan en tipos de datos que representan otros códigos. El complemento [Macro Paradise] [] extiende las capacidades de las macros más allá del lenguaje base. [Paraíso macro]: http://docs.scala-lang.org/overviews/macros/paradise.html

Sintaxis

- def x () = macro x_impl // x es una macro, donde x_impl se usa para transformar el código
- def macroTransform (annottees: Any *): Any = macro impl // Usar en las anotaciones para hacerlas macros

Observaciones

Las macros son una función de idioma que se debe habilitar, ya sea importando `scala.language.macros` o con la opción del compilador `-language:macros` . Solo las definiciones de macros requieren esto; El código que utiliza macros no necesita hacerlo.

Examples

Anotación de macro

Esta simple anotación de macro genera el elemento anotado tal como está.

```
import scala.annotation.{compileTimeOnly, StaticAnnotation}
import scala.reflect.macros.whitebox.Context

@compileTimeOnly("enable macro paradise to expand macro annotations")
class noop extends StaticAnnotation {
  def macroTransform(annottees: Any*): Any = macro linkMacro.impl
}

object linkMacro {
  def impl(c: Context)(annottees: c.Expr[Any]*): c.Expr[Any] = {
    import c.universe._

    c.Expr[Any](q"{..$annottees}")
  }
}
```

El `@compileTimeOnly` anotación genera un error con un mensaje que indica que el `paradise` complemento compilador debe ser incluido para utilizar esta macro. Las instrucciones para incluir

esto a través de SBT están aquí .

Puedes usar la macro definida arriba como esta:

```
@noop
case class Foo(a: String, b: Int)

@noop
object Bar {
  def f(): String = "hello"
}

@noop
def g(): Int = 10
```

Método de macros

Cuando se define un método para que sea una macro, el compilador toma el código que se pasa como su argumento y lo convierte en un AST. A continuación, invoca la implementación de la macro con ese AST y devuelve un nuevo AST que luego se empalma de nuevo a su sitio de llamada.

```
import reflect.macros.blackbox.Context

object Macros {
  // This macro simply sees if the argument is the result of an addition expression.
  // E.g. isAddition(1+1) and isAddition("a"+1).
  // but !isAddition(1+1-1), as the addition is underneath a subtraction, and also
  // !isAddition(x.+), and !isAddition(x.+(a,b)) as there must be exactly one argument.
  def isAddition(x: Any): Boolean = macro isAddition_impl

  // The signature of the macro implementation is the same as the macro definition,
  // but with a new Context parameter, and everything else is wrapped in an Expr.
  def isAddition_impl(c: Context)(expr: c.Expr[Any]): c.Expr[Boolean] = {
    import c.universe._ // The universe contains all the useful methods and types
    val plusName = TermName("+").encodedName // Take the name + and encode it as $plus
    expr.tree match { // Turn expr into an AST representing the code in isAddition(...)
      case Apply(Select(_, `plusName`), List(_)) => reify(true)
      // Pattern match the AST to see whether we have an addition
      // Above we match this AST
      //             Apply (function application)
      //            /     \
      //         Select  List(_) (exactly one argument)
      // (selection ^ of entity, basically the . in x.y)
      //      /          \
      //    _              \
      //               `plusName` (method named +)
      case _                                      => reify(false)
      // reify is a macro you use when writing macros
      // It takes the code given as its argument and creates an Expr out of it
    }
  }
}
```

También es posible tener macros que tomen los `Tree` como argumentos. Por ejemplo, cómo `reify` se utiliza para crear `Expr` s, el `q` (por quasiquote) interpolador cadena nos permite crear y

deconstruir `Tree` s. Tenga en cuenta que podríamos haber usado `q` arriba (`expr.tree` es, sorpresa, un `Tree` sí) también, pero no con propósitos demostrativos.

```
// No Exprs, just Trees
def isAddition_impl(c: Context)(tree: c.Tree): c.Tree = {
  import c.universe._
  tree match {
    // q is a macro too, so it must be used with string literals.
    // It can destructure and create Trees.
    // Note how there was no need to encode + this time, as q is smart enough to do it itself.
    case q"${_} + ${_}" => q"true"
    case _              => q"false"
  }
}
```

Errores en macros

Las macros pueden activar advertencias y errores del compilador mediante el uso de su `Context` .

Digamos que somos particularmente celosos cuando se trata de un código incorrecto, y queremos marcar cada instancia de deuda técnica con un mensaje de información del compilador (no pensemos en lo mala que es esta idea). Podemos usar una macro que no haga nada excepto emitir un mensaje de este tipo.

```
import reflect.macros.blackbox.Context

def debtMark(message: String): Unit = macro debtMark_impl
def debtMarkImpl(c: Context)(message: c.Tree): c.Tree = {
  message match {
    case Literal(Constant(msg: String)) => c.info(c.enclosingPosition, msg, false)
    // false above means "do not force this message to be shown unless -verbose"
    case _                              => c.abort(c.enclosingPosition, "Message must be a
string literal.")
    // Abort causes the compilation to completely fail. It's not even a compile error, where
    // multiple can stack up; this just kills everything.
  }
  q"()" // At runtime this method does nothing, so we return ()
}
```

Además, en lugar de usar `???` Para marcar código no implementado, podemos crear dos macros, `!!!` y `?!?` , que sirven al mismo propósito, pero emiten avisos de compilación. `?!?` hará que se emita una advertencia, y `!!!` causará un error absoluto

```
import reflect.macros.blackbox.Context

def ?!? : Nothing = macro impl_?!?
def !!! : Nothing = macro impl_!!!

def impl_?!?(c: Context): c.Tree = {
  import c.universe._
  c.warning(c.enclosingPosition, "Unimplemented!")
  q"${termNames.ROOTPKG}.scala.Predef.???"
  // If someone were to shadow the scala package, scala.Predef.??? would not work, as it
  // would end up referring to the scala that shadows and not the actual scala.
  // ROOTPKG is the very root of the tree, and acts like it is imported anew in every
```

```
  // expression. It is actually named _root_, but if someone were to shadow it, every
  // reference to it would be an error. It allows us to safely access ??? and know that
  // it is the one we want.
}

def impl_!!!(c: Context): c.Tree = {
  import c.universe._
  c.error(c.enclosingPosition, "Unimplemented!")
  q"${termNames.ROOTPKG}.scala.Predef.???"
}
```

Lea Macros en línea: https://riptutorial.com/es/scala/topic/3808/macros

Capítulo 33: Manejo de errores

Examples

Tratar

Usando Try with `map` , `getOrElse` y `flatMap` :

```
import scala.util.Try

val i = Try("123".toInt)      // Success(123)
i.map(_ + 1).getOrElse(321)   // 124

val j = Try("abc".toInt)      // Failure(java.lang.NumberFormatException)
j.map(_ + 1).getOrElse(321)   // 321

Try("123".toInt) flatMap { i =>
  Try("234".toInt)
    .map(_ + i)
}                             // Success(357)
```

Usando `Try` con el patrón de coincidencia:

```
Try(parsePerson("John Doe")) match {
    case Success(person) => println(person.surname)
    case Failure(ex) => // Handle error ...
}
```

Ya sea

Diferentes tipos de datos para error / éxito

```
def getPersonFromWebService(url: String): Either[String, Person] = {

    val response = webServiceClient.get(url)

    response.webService.status match {
        case 200 => {
            val person = parsePerson(response)
            if(!isValid(person)) Left("Validation failed")
            else Right(person)
        }

        case _ => Left(s"Request failed with error code $response.status")
    }
}
```

Coincidencia de patrones en cualquier valor

```
getPersonFromWebService("http://some-webservice.com/person") match {
    case Left(errorMessage) => println(errorMessage)
```

```
    case Right(person) => println(person.surname)
}
```

Convertir cualquier valor a opción

```
val maybePerson: Option[Person] = getPersonFromWebService("http://some-
webservice.com/person").right.toOption
```

Opción

El uso de valores `null` no se recomienda, a menos que se interactúe con el código Java heredado que se espera que sea `null` . En su lugar, la `Option` debería usarse cuando el resultado de una función puede ser algo (`Some`) o nada (`None`).

Un bloque try-catch es más apropiado para el manejo de errores, pero si la función no puede devolver nada legítimamente, la `Option` es apropiada para el uso y es simple.

Una `Option[T]` puede ser `Some(value)` (contiene un valor de tipo `T`) o `None` :

```
def findPerson(name: String): Option[Person]
```

Si no se encuentra ninguna persona, `None` puede ser devuelta. De lo contrario, se devuelve un objeto de tipo `Some` contienen un objeto `Person` . Lo que sigue son formas de manejar un objeto de tipo `Option` .

La coincidencia de patrones

```
findPerson(personName) match {
    case Some(person) => println(person.surname)
    case None => println(s"No person found with name $personName")
}
```

Usando map y getOrElse

```
val name = findPerson(personName).map(_.firstName).getOrElse("Unknown")
println(name) // Prints either the name of the found person or "Unknown"
```

Utilizando pliegue

```
val name = findPerson(personName).fold("Unknown")(_.firstName)
// equivalent to the map getOrElse example above.
```

Convertir a Java

Si necesita convertir un tipo de `Option` a un tipo de Java con capacidad nula para la

interoperabilidad:

```
val s: Option[String] = Option("hello")
s.orNull           // "hello": String
s.getOrElse(null)  // "hello": String

val n: Option[Int] = Option(42)
n.orNull           // compilation failure (Cannot prove that Null <:< Int.)
n.getOrElse(null)  // 42
```

Manejo de errores originados en futuros

Cuando se lanza una `exception` desde un `Future` , puede (debería) usar `recover` para manejarlo.

Por ejemplo,

```
def runFuture: Future = Future { throw new FairlyStupidException }

val itWillBeAwesome: Future = runFuture
```

... lanzará una `Exception` desde dentro del `Future` . Pero viendo que podemos predecir una `Exception` de tipo `FairlyStupidException` con una alta probabilidad, podemos manejar este caso específicamente de una manera elegante:

```
val itWillBeAwesomeOrIllRecover = runFuture recover {
    case stupid: FairlyStupidException =>
         BadRequest("Another stupid exception!")
}
```

Como puede ver, el método dado para `recover` es un `PartialFunction` sobre el dominio de todos los `Throwable` , por lo que puede manejar solo algunos tipos y luego dejar que el resto entre en el nivel de manejo de excepciones en los niveles más altos en la pila `Future` .

Tenga en cuenta que esto es similar a ejecutar el siguiente código en un contexto no `Future` :

```
def runNotFuture: Unit = throw new FairlyStupidException

try {
    runNotFuture
} catch {
    case e: FairlyStupidException => BadRequest("Another stupid exception!")
}
```

Es realmente importante manejar las excepciones generadas en `Future` s porque la mayoría de las veces son más insidiosas. Por lo general, no se ponen en su lugar porque se ejecutan en un contexto de ejecución y un hilo diferente, y por lo tanto no le piden que los corrija cuando suceden, especialmente si no nota nada en los registros o en el comportamiento de solicitud.

Usando las cláusulas try-catch

Además de las construcciones funcionales, como `Try` , `Option` y `Either` para el manejo de errores,

Scala también admite una sintaxis similar a la de Java, utilizando una cláusula try-catch (con un bloque de potencial también). La cláusula catch es una coincidencia de patrón:

```
try {
  // ... might throw exception
} catch {
  case ioe: IOException => ... // more specific cases first
  case e: Exception => ...
  // uncaught types will be thrown
} finally {
  // ...
}
```

Convertir excepciones en uno u otro tipo de opción

Para convertir excepciones en `Either` o `Option` tipos, puede utilizar métodos que proporcionan en `scala.util.control.Exception`

```
import scala.util.control.Exception._

val plain = "71a"
val optionInt: Option[Int] = catching(classOf[java.lang.NumberFormatException]) opt {
plain.toInt }
val eitherInt = Either[Throwable, Int] = catching(classOf[java.lang.NumberFormatException])
either { plain.toInt }
```

Lea Manejo de errores en línea: https://riptutorial.com/es/scala/topic/910/manejo-de-errores

Capítulo 34: Manejo de XML

Examples

Beautify o Pretty-Print XML

La utilidad `PrettyPrinter` 'imprimirá bastante' documentos XML. El siguiente fragmento de código imprime XML sin formato:

```
import scala.xml.{PrettyPrinter, XML}
val xml = XML.loadString("<a>Alana<b><c>Beth</c><d>Catie</d></b></a>")
val formatted = new PrettyPrinter(150, 4).format(xml)
print(formatted)
```

Esto generará el contenido utilizando un ancho de página de `150` y una constante de sangría de `4` caracteres de espacio en blanco:

```
<a>
    Alana
    <b>
        <c>Beth</c>
        <d>Catie</d>
    </b>
</a>
```

Puede usar `XML.loadFile("nameoffile.xml")` para cargar xml desde un archivo en lugar de desde una cadena.

Lea Manejo de XML en línea: https://riptutorial.com/es/scala/topic/1453/manejo-de-xml

Capítulo 35: Mejores prácticas

Observaciones

> Prefiere vals, objetos inmutables y métodos sin efectos secundarios. Alcanzarlos primero. Use vars, objetos mutables y métodos con efectos secundarios cuando tenga una necesidad específica y una justificación para ellos.

- *Programación en Scala* , por Odersky, Spoon, y Venners

Hay más ejemplos y pautas en esta presentación de Odersky.

Examples

Mantenlo simple

No complique las tareas simples. La mayoría de las veces solo necesitarás:

- tipos de datos algebraicos
- recursion estructural
- Api similar a la mónada (`map` , `map flatMap` , `fold`)

Hay muchas cosas complicadas en Scala, tales como:

- `Cake pattern` o `Reader Monad` para inyección de dependencia.
- Pasando valores arbitrarios como argumentos `implicit` .

Estas cosas no están claras para los recién llegados: evite usarlas antes de que las entienda. El uso de conceptos avanzados sin una necesidad real confunde el código, haciéndolo *menos* fácil de mantener.

No empacar demasiado en una expresión.

- Encuentra nombres significativos para unidades de cómputo.
- Utilice `for` comprensión o `map` para combinar cálculos juntos.

Digamos que tienes algo como esto:

```
if (userAuthorized.nonEmtpy) {
  makeRequest().map {
    case Success(respone) =>
      someProcessing(..)
      if (resendToUser) {
        sendToUser(...)
      }
    ...
  }
}
```

Si todas sus funciones devuelven `Either` u otro tipo de `Validation` , puede escribir:

```
for {
  user     <- authorizeUser
  response <- requestToThirdParty(user)
  _        <- someProcessing(...)
} {
  sendToUser
}
```

Prefiero un estilo funcional, razonablemente

Por defecto:

- Use `val` , no `var` , siempre que sea posible. Esto le permite aprovechar al máximo una serie de utilidades funcionales, incluida la distribución del trabajo.
- Utilice `recursion` y `comprehensions` s, no bucles.
- Utiliza colecciones inmutables. Este es un corolario al uso de `val` siempre que sea posible.
- Concéntrese en las transformaciones de datos, la lógica de estilo CQRS y no CRUD.

Hay buenas razones para elegir el estilo no funcional:

- `var` se puede utilizar para el estado local (por ejemplo, dentro de un actor).
- `mutable` da mejor desempeño en ciertas situaciones.

Lea Mejores prácticas en línea: https://riptutorial.com/es/scala/topic/4376/mejores-practicas

Capítulo 36: Mientras bucles

Sintaxis

- while (boolean_expression) {block_expression}
- do {block_expression} while (boolean_expression)

Parámetros

Parámetro	Detalles
`boolean_expression`	Cualquier expresión que se evalúe como `true` o `false` .
`block_expression`	Cualquier expresión o conjunto de expresiones que se evaluarán si `boolean_expression` evalúa como `true` .

Observaciones

La diferencia principal entre los bucles `while` y `do-while` while es si ejecutan la `block_expression` antes de verificar si deben realizar un bucle.

Debido `do-while` bucles `while` y `do-while` se basan en una expresión para evaluar en `false` para terminar, a menudo requieren que el estado mutable se declare fuera del bucle y luego se modifique dentro del bucle.

Examples

Mientras bucles

```
var line = 0
var maximum_lines = 5

while (line < maximum_lines) {
  line = line + 1
  println("Line number: " + line)
}
```

Do-While Loops

```
var line = 0
var maximum_lines = 5

do {
  line = line + 1
```

```
  println("Line number: " + line)
} while (line < maximum_lines)
```

La `do` / `while` de bucle se utiliza con poca frecuencia en la programación funcional, pero se puede utilizar para evitar la falta de apoyo a la `break` / `continue` constructo, como se ve en otros idiomas:

```
if(initial_condition) do if(filter) {
  ...
} while(continuation_condition)
```

Lea Mientras bucles en línea: https://riptutorial.com/es/scala/topic/650/mientras-bucles

Capítulo 37: Mónadas

Examples

Definición de la mónada

Informalmente, una mónada es un contenedor de elementos, anotados como `F[_]` , empaquetados con 2 funciones: `flatMap` (para transformar este contenedor) y `unit` (para crear este contenedor).

Los ejemplos de bibliotecas comunes incluyen `List[T]` , `Set[T]` y `Option[T]` .

Definicion formal

La mónada `M` es un tipo paramétrico `M[T]` con dos operaciones `flatMap` y `unit` , como:

```
trait M[T] {
  def flatMap[U](f: T => M[U]): M[U]
}

def unit[T](x: T): M[T]
```

Estas funciones deben cumplir tres leyes:

1. *Asociatividad* : `(m flatMap f) flatMap g = m flatMap (x => f(x) flatMap g)`
   Es decir, si la secuencia no se modifica, puede aplicar los términos en cualquier orden. Por lo tanto, aplicar `m` a `f` , y luego aplicar el resultado a `g` dará el mismo resultado que aplicar `f` a `g` , y luego aplicar `m` a ese resultado.
2. *Unidad izquierda* : `unit(x) flatMap f == f(x)`
   Es decir, la unidad de la mónada de `x` asignada de forma plana a través de `f` es equivalente a aplicar `f` a `x` .
3. *Unidad derecha* : `m flatMap unit == m`
   Esta es una "identidad": cualquier mónada con plano mapeado contra unidad devolverá una mónada equivalente a sí misma.

Ejemplo :

```
val m = List(1, 2, 3)
def unit(x: Int): List[Int] = List(x)
def f(x: Int): List[Int] = List(x * x)
def g(x: Int): List[Int] = List(x * x * x)
val x = 1
```

1. *Asociatividad*

```
(m flatMap f).flatMap(g) == m.flatMap(x => f(x) flatMap g) //Boolean = true
//Left side:
List(1, 4, 9).flatMap(g) // List(1, 64, 729)
```

```
//Right side:
 m.flatMap(x => (x * x) * (x * x) * (x * x)) //List(1, 64, 729)
```

2. *Unidad izquierda*

```
unit(x).flatMap(x => f(x)) == f(x)
List(1).flatMap(x => x * x) == 1 * 1
```

3. *Unidad derecha*

```
//m flatMap unit == m
m.flatMap(unit) == m
List(1, 2, 3).flatMap(x => List(x)) == List(1,2,3) //Boolean = true
```

Las colecciones estándar son mónadas

La mayoría de las colecciones estándar son mónadas (`List[T]` , `Option[T]`), o similares a mónadas (`Either[T]` , `Future[T]`). Estas colecciones se pueden combinar fácilmente dentro `for` comprensiones (que son una forma equivalente de escribir transformaciones de `flatMap`):

```
val a = List(1, 2, 3)
val b = List(3, 4, 5)
for {
  i <- a
  j <- b
} yield(i * j)
```

Lo anterior es equivalente a:

```
a flatMap {
  i => b map {
    j => i * j
  }
}
```

Debido a que una mónada conserva la *estructura de datos* y solo actúa sobre los elementos dentro de esa estructura, podemos construir *estructuras de datos* monádicas en cadena sin fin, como se muestra aquí en una para comprensión.

Lea Mónadas en línea: https://riptutorial.com/es/scala/topic/4112/monadas

Capítulo 38: Operadores en Scala

Examples

Operadores incorporados

Scala tiene los siguientes operadores integrados (métodos / elementos de lenguaje con reglas de precedencia predefinidas):

Tipo	Símbolo	Ejemplo
Operadores aritméticos	`+ - * / %`	`a + b`
Operadores relacionales	`== != > < >= <=`	`a > b`
Operadores logicos	`&& & || | !`	`a && b`
Operadores de bits	`& | ^ ~ << >> >>>`	`a & b` , `~a` , `a >>> b`
Operadores de Asignación	`= += -= *= /= %= <<= >>= &= ^= |=`	`a += b`

Los operadores de Scala tienen el mismo significado que en Java.

Nota : los métodos que terminan con `:` enlazar a la derecha (y asociativa a la derecha), por lo que la llamada con `list.::(value)` se puede escribir como `value :: list` con la sintaxis del operador. (`1 :: 2 :: 3 :: Nil` es lo mismo que `1 :: (2 :: (3 :: Nil))`)

Sobrecarga del operador

En Scala puedes definir tus propios operadores:

```
class Team {
   def +(member: Person) = ...
}
```

Con lo definido arriba puedes usarlo como:

```
ITTeam + Jack
```

o

```
ITTeam.+(Jack)
```

Para definir operadores unarios puedes prefijarlo con `unary_` . Por ejemplo, `unary_!`

```
class MyBigInt {
```

```
   def unary_! = ...
}

var a: MyBigInt = new MyBigInt
var b = !a
```

Precedencia del operador

Categoría	Operador	Asociatividad
Sufijo	`() []`	De izquierda a derecha
Unario	`! ~`	De derecha a izquierda
Multiplicativa	`* / %`	De izquierda a derecha
Aditivo	`+ -`	De izquierda a derecha
Cambio	`>> >>> <<`	De izquierda a derecha
Relacional	`> >= < <=`	De izquierda a derecha
Igualdad	`== !=`	De izquierda a derecha
A nivel de bit y	`&`	De izquierda a derecha
Xor bitwise	`^`	De izquierda a derecha
Bitwise o	`|`	De izquierda a derecha
Lógico y	`&&`	De izquierda a derecha
Lógico o	`||`	De izquierda a derecha
Asignación	`= += -= *= /= %= >>= <<= &= ^= |=`	De derecha a izquierda
Coma	`,`	De izquierda a derecha

La programación en Scala ofrece el siguiente esquema basado en el primer carácter del operador. Por ejemplo, `>` es el 1er carácter en el operador `>>>` :

Operador
(todos los demás caracteres especiales)
`* / %`
`+ -`
`:`

Operador
= !
< >
&
^
|
(todas las letras)
(todos los operadores de asignación)

La única excepción a esta regla se refiere a los *operadores de asignación* , por ejemplo, `+=` , `*=` , etc. Si un operador termina con un carácter igual (=) y no es uno de los operadores de comparación `<=` , `>=` , `==` o `!=` , entonces la prioridad del operador es la misma que la asignación simple. En otras palabras, más bajo que el de cualquier otro operador.

Lea Operadores en Scala en línea: https://riptutorial.com/es/scala/topic/6604/operadores-en-scala

Capítulo 39: Paquetes

Introducción

Los paquetes en Scala administran espacios de nombres en grandes programas. Por ejemplo, la `connection` nombre puede ocurrir en los paquetes `com.sql` y `org.http` . Puede usar `com.sql.connection` y `org.http.connection` , respectivamente, para acceder a cada uno de estos paquetes.

Examples

Estructura del paquete

```
package com {
    package utility {
        package serialization {
            class Serializer
            ...
        }
    }
}
```

Paquetes y archivos

La cláusula del paquete no se enlaza directamente con el archivo donde se encuentra. Es posible encontrar elementos comunes de la cláusula del paquete en diferentes archivos. Por ejemplo, las siguientes cláusulas del paquete se encuentran en el archivo math1.scala y en el archivo math2.scala.

Archivo math1.scala

```
package org {
    package math {
        package statistics {
            class Interval
        }
    }
}
```

Archivo math2.scala

```
package org {
    package math{
        package probability {
            class Density
        }
    }
}
```

Archivo study.scala

```
import org.math.probability.Density
import org.math.statistics.Interval

object Study {

    def main(args: Array[String]): Unit = {
        var a = new Interval()
        var b = new Density()
    }
}
```

Paquete de denominación de la convensión

Los paquetes de Scala deben seguir las convenciones de nomenclatura de paquetes Java. Los nombres de paquetes se escriben en minúsculas para evitar conflictos con los nombres de clases o interfaces. Las compañías usan su nombre de dominio de Internet invertido para comenzar los nombres de sus paquetes, por ejemplo,

```
io.super.math
```

Lea Paquetes en línea: https://riptutorial.com/es/scala/topic/8231/paquetes

Capítulo 40: Para expresiones

Sintaxis

- para {cláusulas} cuerpo
- para {cláusulas} cuerpo de rendimiento
- para (cláusulas) cuerpo
- para (cláusulas) ceder cuerpo

Parámetros

Parámetro	Detalles
para	Palabra clave requerida para usar un bucle / comprensión
cláusulas	La iteración y los filtros sobre los que trabaja el.
rendimiento	Use esto si quiere crear o 'producir' una colección. El uso de `yield` hará que el tipo de retorno de `for` sea una colección en lugar de `Unit` .
cuerpo	El cuerpo de la expresión, ejecutado en cada iteración.

Examples

Basic For Loop

```
for (x <- 1 to 10)
  println("Iteration number " + x)
```

Esto demuestra la iteración de una variable, `x` , de `1` a `10` y hacer algo con ese valor. El tipo de retorno de este `for` comprensión es la `Unit` .

Básico Para Comprensión

Esto demuestra un filtro en un bucle for, y el uso del `yield` para crear una 'comprensión de secuencia':

```
for ( x <- 1 to 10 if x % 2 == 0)
  yield x
```

La salida para esto es:

```
scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4, 6, 8, 10)
```

Una para la comprensión es útil cuando necesita crear una nueva colección basada en la iteración y sus filtros.

Anidado para bucle

Esto muestra cómo puedes iterar sobre múltiples variables:

```
for {
  x <- 1 to 2
  y <- 'a' to 'd'
} println("(" + x + "," + y + ")")
```

(Tenga en cuenta que `to` aquí es un método operador infijo que devuelve un rango inclusivo . Véase la definición aquí .)

Esto crea la salida:

```
(1,a)
(1,b)
(1,c)
(1,d)
(2,a)
(2,b)
(2,c)
(2,d)
```

Tenga en cuenta que esta es una expresión equivalente, utilizando paréntesis en lugar de corchetes:

```
for (
  x <- 1 to 2
  y <- 'a' to 'd'
) println("(" + x + "," + y + ")")
```

Para obtener todas las combinaciones en un solo vector, podemos `yield` el resultado y establecerlo en un valor `val` :

```
val a = for {
  x <- 1 to 2
  y <- 'a' to 'd'
} yield "(%s,%s)".format(x, y)
// a: scala.collection.immutable.IndexedSeq[String] = Vector((1,a), (1,b), (1,c), (1,d),
(2,a), (2,b), (2,c), (2,d))
```

Monádico para las comprensiones.

Si tiene varios objetos de tipos monádicos , podemos lograr combinaciones de los valores utilizando un 'para comprensión':

```
for {
   x <- Option(1)
```

```
   y <- Option("b")
   z <- List(3, 4)
} {
    // Now we can use the x, y, z variables
    println(x, y, z)
    x  // the last expression is *not* the output of the block in this case!
}

// This prints
// (1, "b", 3)
// (1, "b", 4)
```

El tipo de retorno de este bloque es la `Unit` .

Si los objetos son del *mismo* tipo monádico `M` (p. Ej., `Option`), usar el `yield` devolverá un objeto de tipo `M` lugar de `Unit` .

```
val a = for {
   x <- Option(1)
   y <- Option("b")
} yield {
    // Now we can use the x, y variables
    println(x, y)
    // whatever is at the end of the block is the output
    (7 * x, y)
}

// This prints:
// (1, "b")
// The val `a` is set:
// a: Option[(Int, String)] = Some((7,b))
```

Tenga en cuenta que la palabra clave de `yield` *no se* puede utilizar en el ejemplo original, donde hay una combinación de tipos monádicos (`Option` y `List`). Tratar de hacerlo producirá un error de falta de coincidencia de tipo de tiempo de compilación.

Iterar a través de colecciones utilizando un bucle for

Esto demuestra cómo imprimir cada elemento de un mapa.

```
val map = Map(1 -> "a", 2 -> "b")
for (number <- map) println(number) // prints (1,a), (2,b)
for ((key, value) <- map) println(value) // prints a, b
```

Esto demuestra cómo imprimir cada elemento de una lista

```
val list = List(1,2,3)
for(number <- list) println(number) // prints 1, 2, 3
```

Desugaring para comprensiones

`for` comprensiones en Scala solo son azúcares sintácticos . Estas comprensiones se

implementan utilizando los `withFilter` , `foreach` , `flatMap` y `map` de sus tipos de materias. Por esta razón, sólo los tipos que han definido estos métodos pueden ser utilizados en una `for` comprensión.

A `for` comprensión de la siguiente forma, con patrones `pN` , generadores `gN` y condiciones `cN` :

```
for(p0 <- x0 if g0; p1 <- g1 if c1) { ??? }
```

... reducirá el azúcar a las llamadas anidadas con `withFilter` y `foreach` :

```
g0.withFilter({ case p0 => c0  case _ => false }).foreach({
  case p0 => g1.withFilter({ case p1 => c1  case _ => false }).foreach({
    case p1 => ???
  })
})
```

Considerando que una expresión `for` / `yield` de la siguiente forma:

```
for(p0 <- g0 if c0; p1 <- g1 if c1) yield ???
```

... reducirá el azúcar a las llamadas anidadas usando `withFilter` y `flatMap` o `map` :

```
g0.withFilter({ case p0 => c0  case _ => false }).flatMap({
  case p0 => g1.withFilter({ case p1 => c1  case _ => false }).map({
    case p1 => ???
  })
})
```

(Tenga en cuenta que el `map` se usa en la comprensión más profunda, y `flatMap` se usa en cada comprensión externa).

A `for` comprensión se puede aplicar a cualquier tipo implementando los métodos requeridos por la representación sin azúcar. No hay restricciones en los tipos de retorno de estos métodos, siempre que sean compositivos.

Lea Para expresiones en línea: https://riptutorial.com/es/scala/topic/669/para-expresiones

Capítulo 41: Programación a nivel de tipo

Examples

Introducción a la programación a nivel de tipo.

Si consideramos una lista heterogénea, en la que los elementos de la lista tienen tipos variados pero conocidos, podría ser deseable poder realizar operaciones en los elementos de la lista colectivamente sin descartar la información de tipo de los elementos. El siguiente ejemplo implementa una operación de mapeo sobre una simple lista heterogénea.

Debido a que el tipo de elemento varía, la clase de operaciones que podemos realizar está restringida a alguna forma de proyección de tipo, por lo que definimos una característica de `Projection` tiene el `type Apply[A]` abstracto `type Apply[A]` calculando el *tipo* de resultado de la proyección, y `def apply[A](a: A): Apply[A]` calculando el *valor* del resultado de la proyección.

```
trait Projection {
  type Apply[A] // <: Any
  def apply[A](a: A): Apply[A]
}
```

En la implementación del `type Apply[A]` , estamos programando en el nivel de tipo (a diferencia del nivel de valor).

Nuestro tipo de lista heterogénea define una operación de `map` parametrizada por la proyección deseada, así como el tipo de proyección. El resultado de la operación del mapa es abstracto, variará según la clase y la proyección, y, naturalmente, debe seguir siendo un `HList` :

```
sealed trait HList {
  type Map[P <: Projection] <: HList
  def map[P <: Projection](p: P): Map[P]
}
```

En el caso de `HNil` , la lista heterogénea vacía, el resultado de cualquier proyección siempre será el mismo. Aquí declaramos el `trait HNil` como una conveniencia para que podamos escribir `HNil` como un tipo en lugar de `HNil.type` :

```
sealed trait HNil extends HList
case object HNil extends HNil {
  type Map[P <: Projection] = HNil
  def map[P <: Projection](p: P): Map[P] = HNil
}
```

`HCons` es la lista heterogénea no vacía. Aquí afirmamos que al aplicar una operación de mapa, el tipo de cabecera resultante es el que resulta de la aplicación de la proyección al valor de cabecera (`P#Apply[H]`), y que el tipo de cola resultante es la que resulta de mapear el proyección sobre la cola (`T#Map[P]`), que se sabe que es una lista `HList` :

```
case class HCons[H, T <: HList](head: H, tail: T) extends HList {
  type Map[P <: Projection] = HCons[P#Apply[H], T#Map[P]]
  def map[P <: Projection](p: P): Map[P] = HCons(p.apply(head), tail.map(p))
}
```

La proyección más obvia es realizar algún tipo de operación de `HCons[Option[String], HCons[Option[Int], HNil]]` siguiente ejemplo proporciona una instancia de `HCons[Option[String], HCons[Option[Int], HNil]]` :

```
HCons("1", HCons(2, HNil)).map(new Projection {
  type Apply[A] = Option[A]
  def apply[A](a: A): Apply[A] = Some(a)
})
```

Lea Programación a nivel de tipo en línea:
https://riptutorial.com/es/scala/topic/3738/programacion-a-nivel-de-tipo

Capítulo 42: Pruebas con ScalaCheck

Introducción

ScalaCheck es una biblioteca escrita en Scala y utilizada para realizar pruebas automatizadas basadas en propiedades de los programas Scala o Java. ScalaCheck se inspiró originalmente en la biblioteca QuickCheck de Haskell, pero también se ha aventurado en su propia cuenta.

ScalaCheck no tiene más dependencias externas que el tiempo de ejecución de Scala, y funciona muy bien con sbt, la herramienta de construcción de Scala. También está completamente integrado en los marcos de prueba ScalaTest y specs2.

Examples

Scalacheck con scalatest y mensajes de error.

Ejemplo de uso de scalacheck con scalatest. A continuación tenemos cuatro pruebas:

- "Mostrar ejemplo de pase" - pasa
- "muestra un ejemplo simple sin mensaje de error personalizado": solo se produjo un mensaje fallido sin detalles, se usa el operador booleano `&&`
- "Mostrar ejemplo con mensajes de error en el argumento": mensaje de error en el argumento (`"argument" |:` :) Se utiliza el método Props.all en lugar de `&&`
- "Mostrar ejemplo con mensajes de error en el comando": mensaje de error en el comando (`"command" |:` :) Se utiliza el método Props.all en lugar de `&&`

```
import org.scalatest.prop.Checkers
import org.scalatest.{Matchers, WordSpecLike}

import org.scalacheck.Gen._
import org.scalacheck.Prop._
import org.scalacheck.Prop

object Splitter {
  def splitLineByColon(message: String): (String, String) = {
    val (command, argument) = message.indexOf(":") match {
      case -1 =>
        (message, "")
      case x: Int =>
        (message.substring(0, x), message.substring(x + 1))
    }
    (command.trim, argument.trim)
  }

  def splitLineByColonWithBugOnCommand(message: String): (String, String) = {
    val (command, argument) = splitLineByColon(message)
    (command.trim + 2, argument.trim)
  }

  def splitLineByColonWithBugOnArgument(message: String): (String, String) = {
```

```
    val (command, argument) = splitLineByColon(message)
    (command.trim, argument.trim + 2)
  }
}

class ScalaCheckSpec extends WordSpecLike with Matchers with Checkers {

  private val COMMAND_LENGTH = 4

  "ScalaCheckSpec " should {
```

```
    "show pass example" in {
      check {
        Prop.forAll(listOfN(COMMAND_LENGTH, alphaChar), alphaStr) {
          (chars, expArgument) =>
            val expCommand = new String(chars.toArray)
            val line = s"$expCommand:$expArgument"
            val (c, p) = Splitter.splitLineByColon(line)
            Prop.all("command" |: c =? expCommand, "argument" |: expArgument =? p)
        }

      }
    }
```

```
"show simple example without custom error message " in {
  check {
    Prop.forAll(listOfN(COMMAND_LENGTH, alphaChar), alphaStr) {
      (chars, expArgument) =>
        val expCommand = new String(chars.toArray)
        val line = s"$expCommand:$expArgument"
        val (c, p) = Splitter.splitLineByColonWithBugOnArgument(line)
        c === expCommand && expArgument === p
    }

  }
}
```

```
"show example with error messages on argument" in {
  check {
    Prop.forAll(listOfN(COMMAND_LENGTH, alphaChar), alphaStr) {
      (chars, expArgument) =>
        val expCommand = new String(chars.toArray)
        val line = s"$expCommand:$expArgument"
        val (c, p) = Splitter.splitLineByColonWithBugOnArgument(line)
        Prop.all("command" |: c =? expCommand, "argument" |: expArgument =? p)
    }

  }
}
```

```
"show example with error messages on command" in {
  check {
    Prop.forAll(listOfN(COMMAND_LENGTH, alphaChar), alphaStr) {
      (chars, expArgument) =>
        val expCommand = new String(chars.toArray)
        val line = s"$expCommand:$expArgument"
        val (c, p) = Splitter.splitLineByColonWithBugOnCommand(line)
        Prop.all("command" |: c =? expCommand, "argument" |: expArgument =? p)
```

```
    }

  }
}
```

La salida (fragmentos):

```
[info] - should show example // passed
[info] - should show simple example without custom error message  *** FAILED ***
[info]    (ScalaCheckSpec.scala:73)
[info]     Falsified after 0 successful property evaluations.
[info]     Location: (ScalaCheckSpec.scala:73)
[info]     Occurred when passed generated values (
[info]       arg0 = List(), // 3 shrinks
[info]       arg1 = ""
[info]     )
[info] - should show example with error messages on argument *** FAILED ***
[info]    (ScalaCheckSpec.scala:86)
[info]     Falsified after 0 successful property evaluations.
[info]     Location: (ScalaCheckSpec.scala:86)
[info]     Occurred when passed generated values (
[info]       arg0 = List(), // 3 shrinks
[info]       arg1 = ""
[info]     )
[info]     Labels of failing property:
[info]       Expected "" but got "2"
[info]       argument
[info] - should show example with error messages on command *** FAILED ***
[info]    (ScalaCheckSpec.scala:99)
[info]     Falsified after 0 successful property evaluations.
[info]     Location: (ScalaCheckSpec.scala:99)
[info]     Occurred when passed generated values (
[info]       arg0 = List(), // 3 shrinks
[info]       arg1 = ""
[info]     )
[info]     Labels of failing property:
[info]       Expected "2" but got ""
[info]       command
```

Lea Pruebas con ScalaCheck en línea: https://riptutorial.com/es/scala/topic/9430/pruebas-con-scalacheck

Capítulo 43: Pruebas con ScalaTest

Examples

Hola World Spec Test

Cree una clase de prueba en el directorio `src/test/scala` , en un archivo llamado `HelloWorldSpec.scala` . Pon esto dentro del archivo:

```
import org.scalatest.{FlatSpec, Matchers}

class HelloWorldSpec extends FlatSpec with Matchers {

  "Hello World" should "not be an empty String" in {
      val helloWorld = "Hello World"
      helloWorld should not be ("")
  }
}
```

- Este ejemplo utiliza `FlatSpec` y `Matchers` , que forman parte de la biblioteca ScalaTest.
- `FlatSpec` permite que las pruebas se escriban en el estilo de **desarrollo impulsado por** el **comportamiento (BDD)** . En este estilo, se usa una oración para describir el comportamiento esperado de una unidad de código dada. La prueba confirma que el código se adhiere a ese comportamiento. Vea la documentación para información adicional .

Hoja de prueba de especificaciones

Preparar

Las siguientes pruebas utilizan estos valores para los ejemplos.

```
val helloWorld = "Hello World"
val helloWorldCount = 1
val helloWorldList = List("Hello World", "Bonjour Le Monde")
def sayHello = throw new IllegalStateException("Hello World Exception")
```

Tipo de verificación

Para verificar el tipo para un determinado `val` :

```
helloWorld shouldBe a [String]
```

Tenga en cuenta que los paréntesis aquí se utilizan para obtener el tipo `String` .

Cheque igual

Para probar la igualdad:

```
helloWorld shouldEqual "Hello World"
helloWorld should === ("Hello World")
helloWorldCount shouldEqual 1
helloWorldCount shouldBe 1
helloWorldList shouldEqual List("Hello World", "Bonjour Le Monde")
helloWorldList === List("Hello World", "Bonjour Le Monde")
```

Cheque no igual

Para probar la desigualdad:

```
helloWorld should not equal "Hello"
helloWorld !== "Hello"
helloWorldCount should not be 5
helloWorldList should not equal List("Hello World")
helloWorldList !== List("Hello World")
helloWorldList should not be empty
```

Verificación de longitud

Para verificar longitud y / o tamaño:

```
helloWorld should have length 11
helloWorldList should have size 2
```

Verificación de excepciones

Para verificar el tipo y mensaje de una excepción:

```
val exception = the [java.lang.IllegalStateException] thrownBy {
  sayHello
}
exception.getClass shouldEqual classOf[java.lang.IllegalStateException]
exception.getMessage should include ("Hello World")
```

Incluir la biblioteca ScalaTest con SBT

Usando SBT para administrar la dependencia de la biblioteca , agregue esto a `build.sbt` :

```
libraryDependencies += "org.scalactic" %% "scalactic" % "3.0.0"
libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.0" % "test"
```

Más detalles se pueden encontrar en el sitio de ScalaTest .

Lea Pruebas con ScalaTest en línea: https://riptutorial.com/es/scala/topic/5506/pruebas-con-scalatest

Capítulo 44: Rasgos

Sintaxis

- rasgo ATrait {...}
- clase AClass (...) extiende ATrait {...}
- clase AClass extiende BClass con ATrait
- clase AClass extiende ATrait con BTrait
- La clase AClass extiende ATrait con BTrait con CTrait.
- clase ATrait extiende BTrait

Examples

Modificación apilable con rasgos

Puede usar rasgos para modificar los métodos de una clase, usando rasgos en forma apilable.

El siguiente ejemplo muestra cómo se pueden apilar los rasgos. El ordenamiento de los rasgos es importante. Usando diferentes orden de rasgos, se logra un comportamiento diferente.

```
class Ball {
  def roll(ball : String) = println("Rolling : " + ball)
}

trait Red extends Ball {
  override def roll(ball : String) = super.roll("Red-" + ball)
}

trait Green extends Ball {
  override def roll(ball : String) = super.roll("Green-" + ball)
}

trait Shiny extends Ball {
  override def roll(ball : String) = super.roll("Shiny-" + ball)
}

object Balls {
  def main(args: Array[String]) {
    val ball1 = new Ball with Shiny with Red
    ball1.roll("Ball-1") // Rolling : Shiny-Red-Ball-1

    val ball2 = new Ball with Green with Shiny
    ball2.roll("Ball-2") // Rolling : Green-Shiny-Ball-2
  }
}
```

Tenga en cuenta que `super` se utiliza para invocar el método `roll()` en ambos rasgos. Solo así podremos lograr una modificación apilable. En casos de modificación apilable, el orden de invocación del método está determinado por la *regla de linealización* .

Fundamentos del rasgo

Esta es la versión más básica de un rasgo en Scala.

```
trait Identifiable {
  def getIdentifier: String
  def printIndentification(): Unit = println(getIdentifier)
}

case class Puppy(id: String, name: String) extends Identifiable {
  def getIdentifier: String = s"$name has id $id"
}
```

Dado que no se declara ninguna `AnyRef` para el carácter `Identifiable` , por defecto se extiende desde la clase `AnyRef` . Debido a que no se proporciona una definición para `getIdentifier` en `Identifiable` , la clase `Puppy` debe implementarla. Sin embargo, `Puppy` hereda la implementación de `printIdentification` de `Identifiable` .

En el REPL:

```
val p = new Puppy("K9", "Rex")
p.getIdentifier  // res0: String = Rex has id K9
p.printIndentification()  // Rex has id K9
```

Resolviendo el problema del diamante

El problema de diamante , o herencia múltiple, es manejado por Scala usando Rasgos, que son similares a las interfaces de Java. Los rasgos son más flexibles que las interfaces y pueden incluir métodos implementados. Esto hace rasgos similares a mixins en otros idiomas.

Scala no admite la herencia de varias clases, pero un usuario puede extender múltiples rasgos en una sola clase:

```
trait traitA {
  def name = println("This is the 'grandparent' trait.")
}

trait traitB extends traitA {
  override def name = {
    println("B is a child of A.")
    super.name
  }

}

trait traitC extends traitA {
  override def name = {
    println("C is a child of A.")
    super.name
  }
}

object grandChild extends traitB with traitC
```

```
grandChild.name
```

Aquí `grandChild` se hereda de `traitB` y `traitC` , que a su vez heredan de `traitA` . La salida (a continuación) también muestra el orden de prioridad al resolver qué implementaciones de métodos se llaman primero:

```
C is a child of A.
B is a child of A.
This is the 'grandparent' trait.
```

Tenga en cuenta que, cuando se utiliza `super` para invocar métodos en `class` o `trait` , la regla de linealización entra en juego para decidir la jerarquía de llamadas. El orden de linealización para `grandChild` será:

> grandChild -> traitC -> traitB -> traitA -> AnyRef -> Any

A continuación se muestra otro ejemplo:

```
trait Printer {
  def print(msg : String) = println (msg)
}

trait DelimitWithHyphen extends Printer {
  override def print(msg : String) {
    println("-------------")
    super.print(msg)
  }
}

trait DelimitWithStar extends Printer  {
  override def print(msg : String) {
    println("*************")
    super.print(msg)
  }
}

class CustomPrinter extends Printer with DelimitWithHyphen with DelimitWithStar

object TestPrinter{
  def main(args: Array[String]) {
    new CustomPrinter().print("Hello World!")
  }
}
```

Este programa imprime:

```
*************
-------------
Hello World!
```

La linealización para `CustomPrinter` será:

> CustomPrinter -> DelimitWithStar -> DelimitWithHyphen -> Printer -> AnyRef -> Any

Linealización

En caso de modificación apilable , Scala organiza clases y rasgos en un orden lineal para determinar la jerarquía de llamadas a métodos, lo que se conoce como *linealización* . La regla de linealización se usa *solo* para aquellos métodos que involucran la invocación de métodos a través de `super()` . Consideremos esto con un ejemplo:

```
class Shape {
  def paint (shape: String): Unit = {
    println(shape)
  }
}

trait Color extends Shape {
  abstract override def paint (shape : String) {
    super.paint(shape + "Color ")
  }
}

trait Blue extends Color {
  abstract override def paint (shape : String) {
    super.paint(shape + "with Blue ")
  }
}

trait Border extends Shape {
  abstract override def paint (shape : String) {
    super.paint(shape + "Border ")
  }
}

trait Dotted extends Border {
  abstract override def paint (shape : String) {
    super.paint(shape + "with Dotted ")
  }
}

class MyShape extends Shape with Dotted with Blue {
  override def paint (shape : String) {
    super.paint(shape)
  }
}
```

La linealización pasa de *atrás hacia delante* . En este caso,

1. La primera `Shape` será linealizada, lo que se parece a:

   ```
   Shape -> AnyRef -> Any
   ```

2. Entonces `Dotted` se linealiza:

   ```
   Dotted -> Border -> Shape -> AnyRef -> Any
   ```

3. El siguiente en la línea es `Blue` . Normalmente la linealización del `Blue` será:

   ```
   Blue -> Color -> Shape -> AnyRef -> Any
   ```

porque, en la linealización de `MyShape` hasta ahora (*Paso 2*), `Shape -> AnyRef -> Any` ya ha aparecido. Por lo tanto, se ignora. Así, la linealización `Blue` será:

```
Blue -> Color -> Dotted -> Border -> Shape -> AnyRef -> Any
```

4. Finalmente, se agregará el `Circle` y el orden de linealización final será:

    Círculo -> Azul -> Color -> Punteado -> Borde -> Forma -> AnyRef -> Cualquiera

Este orden de linealización decide el orden de invocación de los métodos cuando se utiliza `super` en cualquier clase o rasgo. La primera implementación del método desde la derecha se invoca, en el orden de linealización. Si se `new MyShape().paint("Circle ")` , se imprimirá:

```
Circle with Blue Color with Dotted Border
```

Más información sobre la linealización se puede encontrar aquí .

Lea Rasgos en línea: https://riptutorial.com/es/scala/topic/1056/rasgos

Capítulo 45: Recursion

Examples

Recursion de cola

Usando la recursión regular, cada llamada recursiva inserta otra entrada en la pila de llamadas. Cuando se completa la recursión, la aplicación tiene que quitar cada entrada por completo hacia abajo. Si hay muchas llamadas a funciones recursivas, puede terminar con una pila enorme.

Scala elimina automáticamente la recursión en caso de que encuentre la llamada recursiva en la posición de cola. La anotación (`@tailrec`) se puede agregar a las funciones recursivas para garantizar que se realice la optimización de la llamada de cola. El compilador luego muestra un mensaje de error si no puede optimizar su recursión.

Recursion regular

Este ejemplo no es recursivo de cola porque cuando se realiza la llamada recursiva, la función debe realizar un seguimiento de la multiplicación que debe hacer con el resultado después de que se devuelve la llamada.

```
def fact(i : Int) : Int = {
   if(i <= 1) i
   else i * fact(i-1)
}

println(fact(5))
```

La llamada a la función con el parámetro dará como resultado una pila que se verá así:

```
(fact 5)
(* 5 (fact 4))
(* 5 (* 4 (fact 3)))
(* 5 (* 4 (* 3 (fact 2))))
(* 5 (* 4 (* 3 (* 2 (fact 1)))))
(* 5 (* 4 (* 3 (* 2 (* 1 (fact 0))))))
(* 5 (* 4 (* 3 (* 2 (* 1 * 1)))))
(* 5 (* 4 (* 3 (* 2))))
(* 5 (* 4 (* 6)))
(* 5 (* 24))
120
```

Si intentamos anotar este ejemplo con `@tailrec` , obtendremos el siguiente mensaje de error: `could not optimize @tailrec annotated method fact: it contains a recursive call not in tail position`

Recursion de cola

En la recursión de cola, primero realiza sus cálculos y luego ejecuta la llamada recursiva,

pasando los resultados de su paso actual al siguiente paso recursivo.

```
def fact_with_tailrec(i : Int) : Long = {
   @tailrec
   def fact_inside(i : Int, sum: Long) : Long = {
      if(i <= 1) sum
      else fact_inside(i-1,sum*i)
   }
   fact_inside(i,1)
}

println(fact_with_tailrec(5))
```

En contraste, la traza de la pila para el factorial recursivo de la cola se parece a lo siguiente:

```
(fact_with_tailrec 5)
(fact_inside 5 1)
(fact_inside 4 5)
(fact_inside 3 20)
(fact_inside 2 60)
(fact_inside 1 120)
```

Solo existe la necesidad de realizar un seguimiento de la misma cantidad de datos para cada llamada a `fact_inside` porque la función simplemente está devolviendo el valor que llegó a la parte superior. Esto significa que incluso si se llama a `fact_with_tail 1000000` , solo se necesita la misma cantidad de espacio que `fact_with_tail 3` . Este no es el caso con la llamada no recursiva de cola, y como tales valores grandes pueden causar un desbordamiento de pila.

Recursión sin pila con trampolín (scala.util.control.TailCalls)

Es muy común obtener un error `StackOverflowError` al llamar a la función recursiva. La biblioteca estándar de Scala ofrece TailCall para evitar el desbordamiento de pila mediante el uso de objetos de montón y continuaciones para almacenar el estado local de la recursión.

Dos ejemplos del scaladoc de TailCalls

```
import scala.util.control.TailCalls._

def isEven(xs: List[Int]): TailRec[Boolean] =
  if (xs.isEmpty) done(true) else tailcall(isOdd(xs.tail))

def isOdd(xs: List[Int]): TailRec[Boolean] =
  if (xs.isEmpty) done(false) else tailcall(isEven(xs.tail))

// Does this List contain an even number of elements?
isEven((1 to 100000).toList).result

def fib(n: Int): TailRec[Int] =
  if (n < 2) done(n) else for {
    x <- tailcall(fib(n - 1))
    y <- tailcall(fib(n - 2))
  } yield (x + y)

// What is the 40th entry of the Fibonacci series?
```

```
fib(40).result
```

Lea Recursion en línea: https://riptutorial.com/es/scala/topic/3889/recursion

Capítulo 46: Reflexión

Examples

Cargando una clase usando la reflexión

```
import scala.reflect.runtime.universe._
val mirror = runtimeMirror(getClass.getClassLoader)
val module = mirror.staticModule("org.data.TempClass")
```

Lea Reflexión en línea: https://riptutorial.com/es/scala/topic/5824/reflexion

Capítulo 47: Scala.js

Introducción

`Scala.js` es un puerto de `Scala` que se compila a `JavaScript` , que al final se ejecutará fuera de la `JVM` . Tiene ventajas como la escritura fuerte, la optimización de código en tiempo de compilación, la interoperabilidad total con las bibliotecas de JavaScript.

Examples

console.log en Scala.js

```
println("Hello Scala.js") // In ES6: console.log("Hello Scala.js");
```

Funciones de flecha gorda

```
val lastNames = people.map(p => p.lastName)
// Or shorter:
val lastNames = people.map(_.lastName)
```

Clase simple

```
class Person(val firstName: String, val lastName: String) {
  def fullName(): String =
    s"$firstName $lastName"
}
```

Colecciones

```
val personMap = Map(
  10 -> new Person("Roger", "Moore"),
  20 -> new Person("James", "Bond")
)
val names = for {
  (key, person) <- personMap
  if key > 15
} yield s"$key = ${person.firstName}"
```

Manipulando DOM

```
import org.scalajs.dom
import dom.document

def appendP(target: dom.Node, text: String) = {
  val pNode = document.createElement("p")
  val textNode = document.createTextNode(text)
  pNode.appendChild(textNode)
```

```
  target.appendChild(pNode)
}
```

Usando con SBT

Dependencia sbt

```
libraryDependencies += "org.scala-js" %%% "scalajs-dom" % "0.9.1" // (Triple %%%)
```

Corriendo

```
sbt run
```

Corriendo con compilación continua:

```
sbt ~run
```

Compilar en un solo archivo JavaScript:

```
sbt fastOptJS
```

Lea Scala.js en línea: https://riptutorial.com/es/scala/topic/9426/scala-js

Capítulo 48: Scaladoc

Sintaxis

- Va por encima de los métodos, campos, clases o paquetes.
- Comienza con `/**`
- Cada línea tiene un inicio `*` procesando con los comentarios.
- Termina con `*/`

Parámetros

Parámetro	Detalles
Clase especifica	_
`@constructor detail`	Explica el constructor principal de la clase.
Método específico	_
`@return detail`	Detalles sobre lo que se devuelve en el método.
Método, constructor y / o etiquetas de clase.	_
`@param x detail`	Detalles sobre el parámetro de valor `x` en un método o constructor.
`@tparam x detail`	Detalles sobre el parámetro de tipo `x` en un método o constructor.
`@throws detail`	Qué excepciones se pueden lanzar.
Uso	_
`@see detail`	Referencias otras fuentes de información.
`@note detail`	Agrega una nota para las condiciones previas o posteriores, o cualquier otra restricción o expectativa notable.
`@example detail`	Proporciona código de ejemplo o documentación de ejemplo relacionada.
`@usecase detail`	Proporciona una definición de método simplificada para cuando la definición de método completa es demasiado compleja o ruidosa.
Otro	_

Parámetro	Detalles
`@author detail`	Proporciona información sobre el autor de los siguientes.
`@version detail`	Proporciona la versión a la que pertenece esta parte.
`@deprecated detail`	Marca la siguiente entidad como desaprobada.

Examples

Scaladoc simple al método

```
/**
  * Explain briefly what method does here
  * @param x Explain briefly what should be x and how this affects the method.
  * @param y Explain briefly what should be y and how this affects the method.
  * @return Explain what is returned from execution.
  */
def method(x: Int, y: String): Option[Double] = {
  // Method content
}
```

Lea Scaladoc en línea: https://riptutorial.com/es/scala/topic/4518/scaladoc

Capítulo 49: Scalaz

Introducción

Scalaz es una librería Scala para programación funcional.

Proporciona estructuras de datos puramente funcionales para complementar las de la biblioteca estándar de Scala. Define un conjunto de clases de tipos fundamentales (por ejemplo, `Functor` , `Monad`) y las instancias correspondientes para un gran número de estructuras de datos.

Examples

AplicarUso

```
import scalaz._
import Scalaz._

scala> Apply[Option].apply2(some(1), some(2))((a, b) => a + b)
res0: Option[Int] = Some(3)

scala> val intToString: Int => String = _.toString

scala> Apply[Option].ap(1.some)(some(intToString))
res1: Option[String] = Some(1)

scala> Apply[Option].ap(none)(some(intToString))
res2: Option[String] = None

scala> val double: Int => Int = _ * 2

scala> Apply[List].ap(List(1, 2, 3))(List(double))
res3: List[Int] = List(2, 4, 6)

scala> :kind Apply
scalaz.Apply's kind is X[F[A]]
```

FunctorUsage

```
import scalaz._
import Scalaz._
scala> val len: String => Int = _.length
len: String => Int = $$Lambda$1164/96982033@7e758f40

scala> Functor[Option].map(Some("foo"))(len)
res0: Option[Int] = Some(3)

scala> Functor[Option].map(None)(len)
res1: Option[Int] = None

scala> Functor[List].map(List("qwer", "adsfg"))(len)
res2: List[Int] = List(4, 5)
```

```
scala> :kind Functor
scalaz.Functor's kind is X[F[A]]
```

Uso de la flecha

```
import scalaz._
import Scalaz._
scala> val plus1 = (_: Int) + 1
plus1: Int => Int = $$Lambda$1167/11131119649@6a6bfd97

scala> val plus2 = (_: Int) + 2
plus2: Int => Int = $$Lambda$1168/9243295480@6bbe050f

scala> val rev = (_: String).reverse
rev: String => String = $$Lambda$1227/1278001332@72685b74

scala> plus1.first apply (1, "abc")
res0: (Int, String) = (2,abc)

scala> plus1.second apply ("abc", 2)
res1: (String, Int) = (abc,3)

scala> rev.second apply (1, "abc")
res2: (Int, String) = (1,cba)

scala> plus1 *** rev apply(7, "abc")
res3: (Int, String) = (8,cba)

scala> plus1 &&& plus2 apply 7
res4: (Int, Int) = (8,9)

scala> plus1.product apply (1, 2)
res5: (Int, Int) = (2,3)

scala> :kind Arrow
scalaz.Arrow's kind is X[F[A1,A2]]
```

Lea Scalaz en línea: https://riptutorial.com/es/scala/topic/9893/scalaz

Capítulo 50: Si expresiones

Examples

Básico Si Expresiones

En Scala (en contraste con Java y la mayoría de los otros lenguajes), `if` es una **expresión en** lugar de una *declaración* . En cualquier caso, la sintaxis es idéntica:

```
if(x < 1984) {
   println("Good times")
} else if(x == 1984) {
   println("The Orwellian Future begins")
} else {
   println("Poor guy...")
}
```

La implicación de `if` ser una expresión es que puede asignar el resultado de la evalación de la expresión a una variable:

```
val result = if(x > 0) "Greater than 0" else "Less than or equals 0"
\\ result: String = Greater than 0
```

Arriba vemos que la expresión `if` se evalúa y el `result` se establece en ese valor resultante.

El tipo de retorno de una expresión `if` es el **supertipo** de todas las ramas lógicas. Esto significa que para este ejemplo, el tipo de retorno es una `String` . Como no todas las expresiones `if` devuelven un valor (como una instrucción `if` que no tiene `else` lógica de bifurcación), es posible que el tipo de retorno sea `Any` :

```
val result = if(x > 0) "Greater than 0"
// result: Any = Greater than 0
```

Si no se puede devolver ningún valor (por ejemplo, si solo se usan efectos secundarios como `println` dentro de las ramas lógicas), el tipo de retorno será `Unit` :

```
val result = if(x > 0) println("Greater than 0")
// result: Unit = ()
```

`if` expresiones en Scala son similares a cómo funciona el operador ternario en Java . Debido a esta similitud, no hay tal operador en Scala: sería redundante.

> Las llaves se pueden omitir en un `if` la expresión si el contenido es una sola línea.

Lea Si expresiones en línea: https://riptutorial.com/es/scala/topic/4171/si-expresiones

Capítulo 51: Símbolos literales

Observaciones

Scala viene con un concepto de **símbolos** : cadenas que se *internan* , es decir: dos símbolos con el mismo nombre (la misma secuencia de caracteres), en contra de las cadenas, se referirán al mismo objeto durante la ejecución.

Los símbolos son una característica de muchos idiomas: Lisp, Ruby y Erlang y más, sin embargo, en Scala son de uso relativamente pequeño. Buena característica para tener sin embargo.

Utilizar:

Cualquier literal que comience con una comilla simple `'` , seguido de uno o más dígitos, letras o puntuaciones inferiores _ es un símbolo literal. El primer carácter es una excepción ya que no puede ser un dígito.

Buenas definiciones:

```
'ATM
'IPv4
'IPv6
'map_to_operations
'data_format_2006

// Using the `Symbol.apply` method

Symbol("hakuna matata")
Symbol("To be or not to be that is a question")
```

Malas definiciones:

```
'8'th_division
'94_pattern
'bad-format
```

Examples

Reemplazo de cadenas en cláusulas de casos

Digamos que tenemos múltiples fuentes de datos que incluyen *base de datos, archivo, solicitud* y lista de *argumentos* . Dependiendo de la fuente elegida cambiamos nuestro enfoque:

```
def loadData(dataSource: Symbol): Try[String] = dataSource match {
  case 'database => loadDatabase() // Loading data from database
  case 'file =>  loadFile() // Loading data from file
  case 'prompt => askUser() // Asking user for data
  case 'argumentList => argumentListExtract() // Accessing argument list for data
  case _ => Failure(new Exception("Unsupported data source"))
```

```
}
```

Podríamos haber usado muy bien `String` en lugar de `Symbol` . No lo hicimos, porque ninguna de las características de las cadenas es útil en este contexto.

Esto hace que el código sea más simple y menos propenso a errores.

Lea Símbolos literales en línea: https://riptutorial.com/es/scala/topic/6419/simbolos-literales

Capítulo 52: sincronizado

Sintaxis

- `objectToSynchronizeOn.synchronized { /* code to run */}`
- `synchronized {/* code to run, can be suspended with wait */}`

Examples

sincronizar en un objeto

`synchronized` es una construcción de concurrencia de bajo nivel que puede ayudar a evitar que múltiples hilos accedan a los mismos recursos. Introducción a la JVM utilizando el lenguaje Java .

```
anInstance.synchronized {
  // code to run when the intristic lock on `anInstance` is acquired
  // other thread cannot enter concurrently unless `wait` is called on `anInstance` to suspend
  // other threads can continue of the execution of this thread if they `notify` or
`notifyAll` `anInstance`'s lock
}
```

En el caso de los `object` , podría sincronizarse en la clase del objeto, no en la instancia de singleton.

sincronizar implícitamente en este

```
 /* within a class, def, trait or object, but not a constructor */
 synchronized {
   /* code to run when an intrisctic lock on `this` is acquired */
   /* no other thread can get the this lock unless execution is suspended with
    * `wait` on `this`
    */
 }
```

Lea sincronizado en línea: https://riptutorial.com/es/scala/topic/3371/sincronizado

Capítulo 53: Sobrecarga del operador

Examples

Definición de operadores de infijo personalizados

En Scala, los operadores (como `+` , `-` , `*` , `++` , etc.) son solo métodos. Por ejemplo, `1 + 2` se puede escribir como `1.+(2)` . Este tipo de métodos se denominan *'operadores de infijo'* .

Esto significa que los métodos personalizados se pueden definir en sus propios tipos, reutilizando estos operadores:

```
class Matrix(rows: Int, cols: Int, val data: Seq[Seq[Int]]){
  def +(that: Matrix) = {
    val newData = for (r <- 0 until rows) yield
      for (c <- 0 until cols) yield this.data(r)(c) + that.data(r)(c)

    new Matrix(rows, cols, newData)
  }
}
```

Estos operadores definidos como métodos se pueden usar de la siguiente manera:

```
val a = new Matrix(2, 2, Seq(Seq(1,2), Seq(3,4)))
val b = new Matrix(2, 2, Seq(Seq(1,2), Seq(3,4)))

// could also be written a.+(b)
val sum = a + b
```

Tenga en cuenta que los operadores de infijo solo pueden tener un solo argumento; el objeto antes de que el operador llame a su propio operador en el objeto después del operador. Cualquier método Scala con un solo argumento se puede usar como un operador de infijo.

> Esto debe ser usado con parcimony. En general, se considera una buena práctica solo si su propio método hace exactamente lo que uno esperaría de ese operador. En caso de duda, use un nombre más conservador, como `add` lugar de `+` .

Definiendo Operadores Unarios Personalizados

Los operadores unarios se pueden definir precediendo al operador con `unary_` . Los operadores unarios están limitados a `unary_+` , `unary_-` , `unary_!` y `unary_~` :

```
class Matrix(rows: Int, cols: Int, val data: Seq[Seq[Int]]){
  def +(that: Matrix) = {
    val newData = for (r <- 0 until rows) yield
      for (c <- 0 until cols) yield this.data(r)(c) + that.data(r)(c)

    new Matrix(rows, cols, newData)
  }
```

```
  def unary_- = {
    val newData = for (r <- 0 until rows) yield
      for (c <- 0 until cols) yield this.data(r)(c) * -1

    new Matrix(rows, cols, newData)
  }
}
```

El operador unario se puede utilizar de la siguiente manera:

```
val a = new Matrix(2, 2, Seq(Seq(1,2), Seq(3,4)))
val negA = -a
```

> Esto debe ser usado con parcimony. Sobrecargar a un operador unario con una definición que no es lo que uno esperaría puede llevar a la confusión del código.

Lea Sobrecarga del operador en línea: https://riptutorial.com/es/scala/topic/2271/sobrecarga-del-operador

Capítulo 54: Tipo de parametrización (genéricos)

Examples

El tipo de opción

Un buen ejemplo de un tipo parametrizado es el tipo de opción . Esencialmente es solo la siguiente definición (con varios métodos más definidos en el tipo):

```
sealed abstract class Option[+A] {
  def isEmpty: Boolean
  def get: A

  final def fold[B](ifEmpty: => B)(f: A => B): B =
    if (isEmpty) ifEmpty else f(this.get)

  // lots of methods...
}

case class Some[A](value: A) extends Option[A] {
  def isEmpty = false
  def get = value
}

case object None extends Option[Nothing] {
  def isEmpty = true
  def get = throw new NoSuchElementException("None.get")
}
```

También podemos ver que esto tiene un método parametrizado, `fold` , que devuelve algo de tipo `B`

Métodos parametrizados

El tipo de retorno de un método puede depender del *tipo* de parámetro. En este ejemplo, `x` es el parámetro, `A` es el *tipo* de `x` , que se conoce como el *parámetro de tipo* .

```
def f[A](x: A): A = x

f(1)          // 1
f("two")      // "two"
f[Float](3)   // 3.0F
```

Scala utilizará la inferencia de tipos para determinar el tipo de retorno, lo que restringe los métodos a los que se puede llamar en el parámetro. Por lo tanto, se debe tener cuidado: lo siguiente es un error de tiempo de compilación porque `*` no está definido para cada tipo `A` :

```
def g[A](x: A): A = 2 * x  // Won't compile
```

Definiendo la lista de Ints.

```
trait IntList { ... }

class Cons(val head: Int, val tail: IntList) extends IntList { ... }

class Nil extends IntList { ... }
```

¿Pero qué pasa si necesitamos definir la lista de Boolean, Double, etc?

Definiendo lista genérica

```
trait List[T] {
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
}

class Cons[T](val head: [T], val tail: List[T]) extends List[T] {
  def isEmpty: Boolean = false
}

class Nil[T] extends List[T] {
  def isEmpty: Boolean = true

  def head: Nothing = throw NoSuchElementException("Nil.head")

  def tail: Nothing = throw NoSuchElementException("Nil.tail")
}
```

Lea Tipo de parametrización (genéricos) en línea: https://riptutorial.com/es/scala/topic/782/tipo-de-parametrizacion--genericos-

Capítulo 55: Tipos de métodos abstractos únicos (tipos SAM)

Observaciones

Los métodos abstractos únicos son tipos, introducidos en Java 8 , que tienen exactamente un miembro abstracto.

Examples

Sintaxis lambda

NOTA: Esto solo está disponible en Scala 2.12+ (y en las versiones recientes de la versión 2.11.x con los `-Xexperimental -Xfuture` compiler)

Un tipo SAM se puede implementar utilizando un lambda:

2.11.8

```
trait Runnable {
  def run(): Unit
}

val t: Runnable = () => println("foo")
```

El tipo puede tener opcionalmente otros miembros no abstractos:

2.11.8

```
trait Runnable {
  def run(): Unit
  def concrete: Int = 42
}

val t: Runnable = () => println("foo")
```

Lea Tipos de métodos abstractos únicos (tipos SAM) en línea:
https://riptutorial.com/es/scala/topic/3664/tipos-de-metodos-abstractos-unicos--tipos-sam-

Capítulo 56: Trabajando con datos en estilo inmutable.

Observaciones

Los nombres de valores y variables deben estar en la caja inferior del camello

> Los nombres constantes deben estar en la caja superior del camello. Es decir, si el miembro es definitivo, inmutable y pertenece a un objeto de paquete o un objeto, puede considerarse una constante
>
> El método, el valor y los nombres de las variables deben estar en la caja inferior del camello

Fuente: http://docs.scala-lang.org/style/naming-conventions.html

Esta compilación:

```
val (a,b) = (1,2)
// a: Int = 1
// b: Int = 2
```

pero esto no lo hace

```
val (A,B) = (1,2)
// error: not found: value A
// error: not found: value B
```

Examples

No es solo val vs. var.

`val` y `var`

```
scala> val a = 123
a: Int = 123

scala> a = 456
<console>:8: error: reassignment to val
       a = 456

scala> var b = 123
b: Int = 123

scala> b = 321
```

```
b: Int = 321
```

- `val` referencias `val` son inmutables: como una variable `final` en `Java` , una vez que se ha inicializado no se puede cambiar
- `var` referencias `var` son reasignables como una simple declaración de variable en Java

Colecciones inmutables y mutables.

```
  val mut = scala.collection.mutable.Map.empty[String, Int]
  mut += ("123" -> 123)
  mut += ("456" -> 456)
  mut += ("789" -> 789)

  val imm = scala.collection.immutable.Map.empty[String, Int]
  imm + ("123" -> 123)
  imm + ("456" -> 456)
  imm + ("789" -> 789)

  scala> mut
    Map(123 -> 123, 456 -> 456, 789 -> 789)

  scala> imm
    Map()

scala> imm + ("123" -> 123) + ("456" -> 456) + ("789" -> 789)
    Map(123 -> 123, 456 -> 456, 789 -> 789)
```

La biblioteca estándar de Scala ofrece estructuras de datos inmutables y mutables, no la referencia a ella. Cada vez que una estructura de datos inmutables se "modifica", se produce una nueva instancia en lugar de modificar la colección original en el lugar. Cada instancia de la colección puede compartir una estructura significativa con otra instancia.

Colección mutable e inmutable (Documentación Oficial Scala)

¡Pero no puedo usar la inmutabilidad en este caso!

Tomemos como ejemplo una función que toma 2 `Map` y devolvemos un `Map` contiene cada elemento en `ma` y `mb` :

```
def merge2Maps(ma: Map[String, Int], mb: Map[String, Int]): Map[String, Int]
```

Un primer intento podría ser iterar a través de los elementos de uno de los mapas usando `for ((k, v) <- map)` y de alguna manera devolver el mapa combinado.

```
def merge2Maps(ma: ..., mb: ...): Map[String, Int] = {

  for ((k, v) <- mb) {
    ???
  }

}
```

Este primer movimiento agrega inmediatamente una restricción: **ahora se *necesita* una mutación fuera de la misma `for`** . Esto es más claro cuando se elimina el azúcar `for` :

```
// this:
for ((k, v) <- map) { ??? }

// is equivalent to:
map.foreach { case (k, v) => ??? }
```

"¿Por qué tenemos que mutar?"

`foreach` basa en los efectos secundarios. Cada vez que queremos que algo suceda dentro de un `foreach` necesitamos "efectos secundarios", en este caso podríamos mutar un `var result` variable `var result` o podemos usar una estructura de datos mutable.

Creando y rellenando el mapa de `result`

Supongamos que `ma` y `mb` son `scala.collection.immutable.Map` , podríamos crear el Mapa de `result` de `ma` :

```
val result = mutable.Map() ++ ma
```

Luego itere a través de `mb` agregando sus elementos y si la `key` del elemento actual en `ma` ya existe, anulémosla con `mb` one.

```
mb.foreach { case (k, v) => result += (k -> v) }
```

Implementación mutable

Hasta ahora todo bien, "tuvimos que usar colecciones mutables" y una implementación correcta podría ser:

```
def merge2Maps(ma: Map[String, Int], mb: Map[String, Int]): Map[String, Int] = {
  val result = scala.collection.mutable.Map() ++ ma
  mb.foreach { case (k, v) => result += (k -> v) }
  result.toMap // to get back an immutable Map
}
```

Como se esperaba:

```
scala> merge2Maps(Map("a" -> 11, "b" -> 12), Map("b" -> 22, "c" -> 23))
  Map(a -> 11, b -> 22, c -> 23)
```

Plegado al rescate.

¿Cómo podemos deshacernos de `foreach` en este escenario? Si lo único que debemos hacer es iterar básicamente sobre los elementos de la colección y aplicar una función mientras se acumula el resultado en la opción, podría estar usando `.foldLeft` :

```
def merge2Maps(ma: Map[String, Int], mb: Map[String, Int]): Map[String, Int] = {
  mb.foldLeft(ma) { case (result, (k, v)) => result + (k -> v) }
  // or more concisely mb.foldLeft(ma) { _ + _ }
}
```

En este caso, nuestro "resultado" es el valor acumulado a partir de `ma` , el `zero` de `.foldLeft` .

Resultado intermedio

Obviamente, esta solución inmutable produce y destruye muchas instancias del `Map` mientras se pliega, pero vale la pena mencionar que esas instancias no son un clon completo del `Map` acumulado, sino que comparten una estructura significativa (datos) con la instancia existente.

Razonabilidad más fácil

Es más fácil razonar acerca de la semántica si es más declarativo como el enfoque `.foldLeft` . El uso de estructuras de datos inmutables podría ayudar a hacer que nuestra implementación sea más fácil de razonar.

Lea Trabajando con datos en estilo inmutable. en línea:
https://riptutorial.com/es/scala/topic/6298/trabajando-con-datos-en-estilo-inmutable-

Capítulo 57: Trabajando con gradle

Examples

Configuración básica

1. Cree un archivo llamado `SCALA_PROJECT/build.gradle` con estos contenidos:

```
group 'scala_gradle'
version '1.0-SNAPSHOT'

apply plugin: 'scala'

repositories {
    jcenter()
    mavenCentral()
    maven {
        url "https://repo.typesafe.com/typesafe/maven-releases"
    }
}

dependencies {
    compile group: 'org.scala-lang', name: 'scala-library', version: '2.10.6'
}


task  "create-dirs" << {
    sourceSets*.scala.srcDirs*.each { it.mkdirs() }
    sourceSets*.resources.srcDirs*.each { it.mkdirs() }
}
```

2. Ejecutar `gradle tasks` para ver las tareas disponibles.

3. Ejecute `gradle create-dirs` para crear un directorio `src/scala, src/resources` .

4. Ejecute `gradle build` para compilar el proyecto y descargar dependencias.

Crea tu propio complemento Gradle Scala

Después de pasar por el ejemplo de la **Configuración básica** , es posible que repita la mayor parte de él en cada uno de los proyectos de Scala Gradle. Huele a código repetitivo ...

¿Qué pasaría si, en lugar de aplicar el complemento de Scala ofrecido por Gradle, pudiera aplicar su propio complemento de Scala, que sería responsable de manejar toda su lógica de compilación común, extendiendo, al mismo tiempo, el complemento ya existente.

Este ejemplo va a transformar la lógica de compilación anterior en un complemento de Gradle reutilizable.

Afortunadamente, en Gradle, puede escribir fácilmente complementos personalizados con la

ayuda de la API de Gradle, como se describe en la documentación . Como lenguaje de implementación, puede usar Scala en sí o incluso Java. Sin embargo, la mayoría de los ejemplos que puede encontrar en todos los documentos están escritos en Groovy. Si necesita más muestras de código o quiere comprender qué hay detrás del complemento de Scala, por ejemplo, puede consultar el repositorio de github de Gradle .

Escribiendo el plugin

Requerimientos

El complemento personalizado agregará la siguiente funcionalidad cuando se aplique a un proyecto:

- un objeto de propiedad `scalaVersion` , que tendrá dos propiedades predeterminadas reemplazables
  - major = "2.12"
  - menor = "0"
- una función `withScalaVersion` , que se aplica a un nombre de dependencia, agregará la versión principal de scala para garantizar la compatibilidad binaria (el operador sbt `%%` podría sonar una campana, de lo contrario, vaya aquí antes de continuar)
- una tarea `createDirs` para crear el árbol de directorios necesario, exactamente como en el ejemplo anterior

Pauta de implementación

1. cree un nuevo proyecto de Gradle y agregue lo siguiente a `build.gradle`

```
apply plugin: 'scala'
apply plugin: 'maven'

repositories {
    mavenLocal()
    mavenCentral()
}

dependencies {
    compile gradleApi()
    compile "org.scala-lang:scala-library:2.12.0"
}
```

Notas :

- La implementación del complemento está escrita en Scala, por lo que necesitamos el complemento Gradle Scala.
- para utilizar el complemento de otros proyectos, se utiliza el complemento Gradle Maven; esto agrega la tarea de `install` utilizada para guardar el archivo jar en el repositorio local de Maven
- `compile gradleApi()` agrega el `gradle-api-<gradle_version>.jar` a la ruta de `gradle-api-<gradle_version>.jar`

2. Crear una nueva clase Scala para la implementación del complemento.

```
package com.btesila.gradle.plugins

import org.gradle.api.{Plugin, Project}

class ScalaCustomPlugin extends Plugin[Project] {
    override def apply(project: Project): Unit = {
        project.getPlugins.apply("scala")
    }
}
```

Notas :

- para implementar un complemento, simplemente extienda el rasgo de `Plugin` del tipo `Project` y anule el método de `apply`
- dentro del método de aplicación, tiene acceso a la instancia de `Project` a la que se aplica el complemento y puede usarlo para agregarle lógica de compilación
- este complemento no hace más que aplicar el ya existente Gradle Scala Plugin

3. agregar la propiedad del objeto `scalaVersion`

En primer lugar, creamos una clase `ScalaVersion` , que contendrá las dos propiedades de la versión

```
class ScalaVersion {
  var major: String = "2.12"
  var minor: String = "0"
}
```

Una cosa interesante acerca de los complementos de Gradle es el hecho de que siempre puede agregar o anular propiedades específicas. Un complemento recibe este tipo de información del usuario a través del `ExtensionContainer` adjunto a una instancia del `Project` gradle. Para más detalles, mira esto .
Al agregar lo siguiente al método de `apply` , básicamente estamos haciendo esto:

- Si no hay una propiedad `scalaVersion` definida en el proyecto, agregamos una con los valores predeterminados.
- de lo contrario, obtenemos el existente como instancia de `ScalaVersion` , para usarlo aún más

```
var scalaVersion = new ScalaVersion
if (!project.getExtensions.getExtraProperties.has("scalaVersion"))
   project.getExtensions.getExtraProperties.set("scalaVersion", scalaVersion)
else
   scalaVersion =
project.getExtensions.getExtraProperties.get("scalaVersion").asInstanceOf[ScalaVersion]
```

Esto es equivalente a escribir lo siguiente en el archivo de compilación del proyecto que aplica el complemento:

```
ext {
    scalaVersion.major = "2.12"
    scalaVersion.minor = "0"

}
```

4. agregue la biblioteca `scala-lang` a las dependencias del proyecto, usando `scalaVersion`

```
project.getDependencies.add("compile", s"org.scala-lang:scala-
library:${scalaVersion.major}.${scalaVersion.minor}")
```

Esto es equivalente a escribir lo siguiente en el archivo de compilación del proyecto que aplica el complemento:

```
compile "org.scala-lang:scala-library:2.12.0"
```

5. agrega la función `withScalaVersion`

```
val withScalaVersion = (lib: String) => {
    val libComp = lib.split(":")
    libComp.update(1, s"${libComp(1)}_${scalaVersion.major}")
    libComp.mkString(":")
}
project.getExtensions.getExtraProperties.set("withScalaVersion", withScalaVersion)
```

6. Finalmente, cree la tarea `createDirs` y agréguela al proyecto.
   Implementar una tarea de Gradle extendiendo `DefaultTask` :

```
class CreateDirs extends DefaultTask {
  @TaskAction
  def createDirs(): Unit = {
    val sourceSetContainer =
this.getProject.getConvention.getPlugin(classOf[JavaPluginConvention]).getSourceSets

    sourceSetContainer forEach { sourceSet =>
      sourceSet.getAllSource.getSrcDirs.forEach(file => if (!file.getName.contains("java"))
file.mkdirs())
    }
  }
}
```

Nota : el `SourceSetContainer` tiene información sobre todos los directorios de origen presentes en el proyecto. Lo que hace el complemento Gradle Scala es agregar los conjuntos de fuente adicionales a los de Java, como se puede ver en los documentos del complemento .

Agregue la tarea `createDir` al proyecto agregando esto al método de `apply` :

```
project.getTasks.create("createDirs", classOf[CreateDirs])
```

Al final, su clase `ScalaCustomPlugin` debería verse así:

```
class ScalaCustomPlugin extends Plugin[Project] {
  override def apply(project: Project): Unit = {
    project.getPlugins.apply("scala")

    var scalaVersion = new ScalaVersion
    if (!project.getExtensions.getExtraProperties.has("scalaVersion"))
      project.getExtensions.getExtraProperties.set("scalaVersion", scalaVersion)
    else
      scalaVersion =
project.getExtensions.getExtraProperties.get("scalaVersion").asInstanceOf[ScalaVersion]

    project.getDependencies.add("compile", s"org.scala-lang:scala-
library:${scalaVersion.major}.${scalaVersion.minor}")

    val withScalaVersion = (lib: String) => {
      val libComp = lib.split(":")
      libComp.update(1, s"${libComp(1)}_${scalaVersion.major}")
      libComp.mkString(":")
    }
    project.getExtensions.getExtraProperties.set("withScalaVersion", withScalaVersion)

    project.getTasks.create("createDirs", classOf[CreateDirs])
  }
}
```

Instalar el proyecto de plugin en el repositorio local de Maven

Esto se hace realmente fácil ejecutando `gradle install`
Puede verificar la instalación yendo al directorio del repositorio local, que generalmente se encuentra en `~/.m2/repository`

¿Cómo encuentra Gradle nuestro nuevo plugin?

Cada complemento de Gradle tiene un `id` que se utiliza en la declaración de `apply` . Por ejemplo, al escribir lo siguiente en el archivo de compilación, se traduce en un desencadenante para Gradle para encontrar y aplicar el complemento con id `scala` .

```
apply plugin: 'scala'
```

De la misma manera, nos gustaría aplicar nuestro nuevo complemento de la siguiente manera,

```
apply plugin: "com.btesila.scala.plugin"
```

lo que significa que nuestro complemento tendrá el id `com.btesila.scala.plugin` .

Para establecer este ID, agregue el siguiente archivo:
src / main / resources / META-INF / gradle-plugin / com.btesil.scala.plugin.properties

```
implementation-class=com.btesila.gradle.plugins.ScalaCustomPlugin
```

Después, ejecute de nuevo la `gradle install` .

Usando el plugin

1. cree un nuevo proyecto Gradle vacío y agregue lo siguiente al archivo de compilación

```
buildscript {
    repositories {
        mavenLocal()
        mavenCentral()
    }

    dependencies {
        //modify this path to match the installed plugin project in your local repository
        classpath 'com.btesila:working-with-gradle:1.0-SNAPSHOT'
    }
}

repositories {
    mavenLocal()
    mavenCentral()
}


apply plugin: "com.btesila.scala.plugin"
```

2. ejecute `gradle createDirs` - ahora debería tener todos los directorios de origen generados
3. reemplaza la versión de Scala agregando esto al archivo de compilación:

```
ext {
    scalaVersion.major = "2.11"
    scalaVersion.minor = "8"

}
println(project.ext.scalaVersion.major)
println(project.ext.scalaVersion.minor)
```

4. agregue una biblioteca de dependencias que sea compatible con los binarios de la versión Scala

```
dependencies {
    compile withScalaVersion("com.typesafe.scala-logging:scala-logging:3.5.0")
}
```

¡Eso es! Ahora puede usar este complemento en todos sus proyectos sin repetir el mismo texto tradicional.

Lea Trabajando con gradle en línea: https://riptutorial.com/es/scala/topic/3304/trabajando-con-gradle

Capítulo 58: Tuplas

Observaciones

¿Por qué las tuplas se limitan a la longitud 23?

Las tuplas son reescritas como objetos por el compilador. El compilador tiene acceso a `Tuple1` través de `Tuple22` . Este límite arbitrario fue decidido por los diseñadores de idiomas.

¿Por qué las longitudes de la tupla cuentan desde 0?

Un `Tuple0` es equivalente a una `Unit` .

Examples

Creando un nuevo tuple

Una tupla es una colección heterogénea de dos a veintidós valores. Una tupla se puede definir utilizando paréntesis. Para tuplas de tamaño `2` (también llamadas 'par') hay una sintaxis de flecha.

```
scala> val x = (1, "hello")
x: (Int, String) = (1,hello)
scala> val y = 2 -> "world"
y: (Int, String) = (2,world)
scala> val z = 3 → "foo"     //example of using U+2192 RIGHTWARD ARROW
z: (Int, String) = (3,foo)
```

`x` es una tupla de tamaño dos. Para acceder a los elementos de una tupla use `._1` , a través de `._22` . Por ejemplo, podemos usar `x._1` para acceder al primer elemento de la tupla `x` . `x._2` accede al segundo elemento. Más elegantemente, puedes usar extractores de tuplas .

La sintaxis de la flecha para crear tuplas de tamaño dos se usa principalmente en Mapas, que son colecciones de pares `(key -> value)` :

```
scala> val m = Map[Int, String](2 -> "world")
m: scala.collection.immutable.Map[Int,String] = Map(2 -> world)

scala> m + x
res0: scala.collection.immutable.Map[Int,String] = Map(2 -> world, 1 -> hello)

scala> (m + x).toList
res1: List[(Int, String)] = List((2,world), (1,hello))
```

La sintaxis para el par en el mapa es la sintaxis de la flecha, dejando claro que 1 es la clave y a es el valor asociado con esa clave.

Tuplas dentro de las colecciones

Las tuplas se usan a menudo dentro de las colecciones, pero deben manejarse de una manera específica. Por ejemplo, dada la siguiente lista de tuplas:

```
scala> val l = List(1 -> 2, 2 -> 3, 3 -> 4)
l: List[(Int, Int)] = List((1,2), (2,3), (3,4))
```

Puede parecer natural agregar los elementos juntos mediante el desempaquetado implícito de la tupla:

```
scala> l.map((e1: Int, e2: Int) => e1 + e2)
```

Sin embargo, esto resulta en el siguiente error:

```
<console>:9: error: type mismatch;
 found   : (Int, Int) => Int
 required: ((Int, Int)) => ?
              l.map((e1: Int, e2: Int) => e1 + e2)
```

Scala no puede desempaquetar implícitamente las tuplas de esta manera. Tenemos dos opciones para arreglar este mapa. El primero es usar los `_1` posicionales `_1` y `_2` :

```
scala> l.map(e => e._1 + e._2)
res1: List[Int] = List(3, 5, 7)
```

La otra opción es usar una declaración de `case` para descomprimir las tuplas usando un patrón de coincidencia:

```
scala> l.map{ case (e1: Int, e2: Int) => e1 + e2}
res2: List[Int] = List(3, 5, 7)
```

Estas restricciones se aplican a cualquier función de orden superior aplicada a una colección de tuplas.

Lea Tuplas en línea: https://riptutorial.com/es/scala/topic/4971/tuplas

Capítulo 59: Unidades de manipulación (medidas)

Sintaxis

- Medidor de clase (medidores de val: doble) extiende AnyVal
- tipo metro = doble

Observaciones

Se recomienda usar clases de valor para las unidades o una biblioteca dedicada para ellas.

Examples

Tipo de alias

```
type Meter = Double
```

Este enfoque simple tiene serios inconvenientes para el manejo de la unidad, ya que cualquier otro tipo que sea `Double` será compatible con él:

```
type Second = Double
var length: Meter = 3
val duration: Second = 1
length = duration
length = 0d
```

Todas las compilaciones anteriores, por lo que en este caso las unidades solo se pueden usar para marcar los tipos de entrada / salida para los lectores del código (solo la intención).

Clases de valor

```
case class Meter(meters: Double) extends AnyVal
case class Gram(grams: Double) extends AnyVal
```

Las clases de valor proporcionan una forma segura para el tipo de codificación de unidades, incluso si requieren un poco más de caracteres para usarlas:

```
var length = Meter(3)
var weight = Gram(4)
//length = weight //type mismatch; found : Gram required: Meter
```

Al extender `AnyVal` s, no hay una penalización de tiempo de ejecución por usarlos, en el nivel JVM, esos son tipos primitivos regulares (`Double` s en este caso).

En caso de que desee generar automáticamente otras unidades (como `Velocity` aka `MeterPerSecond`), este enfoque no es el mejor, aunque hay bibliotecas que también se pueden usar en esos casos:

- Squants
- unidades
- ScalaQuantity

Lea Unidades de manipulación (medidas) en línea:
https://riptutorial.com/es/scala/topic/5966/unidades-de-manipulacion--medidas-

Capítulo 60: Var, Val y Def

Observaciones

Como los `val` son semánticamente estáticos, se inicializan "en el lugar" donde aparecen en el código. Esto puede producir un comportamiento sorprendente e indeseable cuando se usa en clases abstractas y rasgos.

Por ejemplo, digamos que nos gustaría hacer un rasgo llamado `PlusOne` que defina una operación de incremento en un `Int` envuelto. Dado que los `Int` s son inmutables, el valor más uno se conoce en la inicialización y nunca se cambiará después, por lo que semánticamente es un valor `val` . Sin embargo, definirlo de esta manera producirá un resultado inesperado.

```
trait PlusOne {
    val i:Int

    val incr = i + 1
}

class IntWrapper(val i: Int) extends PlusOne
```

No importa cuál es el valor `i` se construye `IntWrapper` con, llamando `.incr` en el objeto devuelto siempre devuelve 1. Esto es porque el val `incr` es inicializado *en el rasgo,* antes de la clase que se extiende, y en ese momento `i` sólo tiene el valor por defecto de `0` . (En otras condiciones, puede completarse con `Nil` , `null` o un valor predeterminado similar).

La regla general, entonces, es evitar usar `val` en cualquier valor que dependa de un campo abstracto. En su lugar, use `lazy val` , que no evalúa hasta que se necesita, o `def` , que evalúa cada vez que se llama. Sin embargo, tenga en cuenta que si el valor de `lazy val` es forzado a evaluar por un `val` antes de que se complete la inicialización, ocurrirá el mismo error.

Un violín (escrito en Scala-Js, pero se aplica el mismo comportamiento) se puede encontrar aquí.

Examples

Var, Val y Def

var

Una `var` es una variable de referencia, similar a las variables en lenguajes como Java. Se pueden asignar diferentes objetos a una `var` libremente, siempre que el objeto dado tenga el mismo tipo con el que se declaró la `var` :

```
scala> var x = 1
x: Int = 1
```

```
scala> x = 2
x: Int = 2

scala> x = "foo bar"
<console>:12: error: type mismatch;
 found   : String("foo bar")
 required: Int
       x = "foo bar"
       ^
```

Observe en el ejemplo anterior el tipo de la `var` fue inferida por el compilador dada la primera asignación de valor.

val

Un `val` es una referencia constante. Por lo tanto, no se puede asignar un nuevo objeto a un `val` que ya se ha asignado.

```
scala> val y = 1
y: Int = 1

scala> y = 2
<console>:12: error: reassignment to val
       y = 2
         ^
```

Sin embargo, el objeto al que apunta un `val` *no* es constante. Ese objeto puede ser modificado:

```
scala> val arr = new Array[Int](2)
arr: Array[Int] = Array(0, 0)

scala> arr(0) = 1

scala> arr
res1: Array[Int] = Array(1, 0)
```

def

Una `def` define un método. Un método no puede ser reasignado a

```
scala> def z = 1
z: Int

scala> z = 2
<console>:12: error: value z_= is not a member of object $iw
       z = 2
       ^
```

En los ejemplos anteriores, `val y` y `def z` devuelven el mismo valor. Sin embargo, una `def` se evalúa *cuando se llama* , mientras que una `val` o `var` se evalúa *cuando se asigna* . Esto puede

resultar en un comportamiento diferente cuando la definición tiene efectos secundarios:

```
scala> val a = {println("Hi"); 1}
Hi
a: Int = 1

scala> def b = {println("Hi"); 1}
b: Int

scala> a + 1
res2: Int = 2

scala> b + 1
Hi
res3: Int = 2
```

Funciones

Como las funciones son valores, se pueden asignar a `val` / `var` / `def` s. Todo lo demás funciona de la misma manera que arriba:

```
scala> val x = (x: Int) => s"value=$x"
x: Int => String = <function1>

scala> var y = (x: Int) => s"value=$x"
y: Int => String = <function1>

scala> def z = (x: Int) => s"value=$x"
z: Int => String

scala> x(1)
res0: String = value=1

scala> y(2)
res1: String = value=2

scala> z(3)
res2: String = value=3
```

Perezoso val

`lazy val` es una función de lenguaje en la que la inicialización de un `val` se retrasa hasta que se accede por primera vez. Después de ese punto, actúa como un `val` normal.

Para usarlo agregue la palabra clave `lazy` antes de `val` . Por ejemplo, usando el REPL:

```
scala> lazy val foo = {
     |   println("Initializing")
     |   "my foo value"
     | }
foo: String = <lazy>

scala> val bar = {
     |   println("Initializing bar")
```

```
     |   "my bar value"
     | }
Initializing bar
bar: String = my bar value

scala> foo
Initializing
res3: String = my foo value

scala> bar
res4: String = my bar value

scala> foo
res5: String = my foo value
```

Este ejemplo demuestra el orden de ejecución. Cuando el `lazy val` se declara, todo lo que se guarda en el `foo` valor es una llamada a la función vago que no ha sido evaluado. Cuando el normal `val` se establece, vemos el `println` llamada de ejecutar y se le asigna el valor a `bar` . Cuando evaluamos `foo` la primera vez, vemos `println` ejecuta `println` , pero no cuando se evalúa la segunda vez. De manera similar, cuando se evalúa la `bar` , no vemos que se ejecute `println` , solo cuando se declara.

Cuando usar 'perezoso'

1. **La inicialización es computacionalmente costosa y el uso de `val` es raro.**

   ```
   lazy val tiresomeValue = {(1 to 1000000).filter(x => x % 113 == 0).sum}
   if (scala.util.Random.nextInt > 1000) {
     println(tiresomeValue)
   }
   ```

   `tiresomeValue` tarda mucho tiempo en calcularse, y no siempre se usa. Si se convierte en un valor `lazy val` ahorra cómputo innecesario.

2. **Resolución de dependencias cíclicas.**

   Veamos un ejemplo con dos objetos que deben declararse al mismo tiempo durante la creación de instancias:

   ```
   object comicBook {
     def main(args:Array[String]): Unit = {
       gotham.hero.talk()
       gotham.villain.talk()
     }
   }

   class Superhero(val name: String) {
     lazy val toLockUp = gotham.villain
     def talk(): Unit = {
       println(s"I won't let you win ${toLockUp.name}!")
     }
   }
   ```

```
class Supervillain(val name: String) {
  lazy val toKill = gotham.hero
  def talk(): Unit = {
    println(s"Let me loosen up Gotham a little bit ${toKill.name}!")
  }
}

object gotham {
  val hero: Superhero = new Superhero("Batman")
  val villain: Supervillain = new Supervillain("Joker")
}
```

Sin la palabra clave `lazy` , los objetos respectivos no pueden ser miembros de un objeto. La ejecución de dicho programa daría lugar a una `java.lang.NullPointerException` . Mediante el uso `lazy` , la referencia se puede asignar antes de que se inicialice, sin temor a tener un valor sin inicializar.

Sobrecarga def

Puede sobrecargar una `def` si la firma es diferente:

```
def printValue(x: Int) {
  println(s"My value is an integer equal to $x")
}

def printValue(x: String) {
  println(s"My value is a string equal to '$x'")
}

printValue(1)  // prints "My value is an integer equal to 1"
printValue("1") // prints "My value is a string equal to '1'"
```

Esto funciona de la misma manera ya sea dentro de clases, rasgos, objetos o no.

Parámetros con nombre

Al invocar una `def` , los parámetros pueden asignarse explícitamente por nombre. Si lo hace, significa que no necesitan ser ordenados correctamente. Por ejemplo, defina `printUs()` como:

```
// print out the three arguments in order.
def printUs(one: String, two: String, three: String) =
   println(s"$one, $two, $three")
```

Ahora se puede llamar de estas maneras (entre otras):

```
printUs("one", "two", "three")
printUs(one="one", two="two", three="three")
printUs("one", two="two", three="three")
printUs(three="three", one="one", two="two")
```

Esto hace que se impriman `one, two, three` en todos los casos.

Si no se nombran todos los argumentos, los primeros argumentos se comparan por orden. Ningún argumento posicional (sin nombre) puede seguir a uno nombrado:

```
printUs("one", two="two", three="three") // prints 'one, two, three'
printUs(two="two", three="three", "one") // fails to compile: 'positional after named
argument'
```

Lea Var, Val y Def en línea: https://riptutorial.com/es/scala/topic/3155/var--val-y-def

Capítulo 61: Variación de tipo

Examples

Covarianza

El símbolo `+` marca un parámetro de tipo como *covariante* . Aquí decimos que "El `Producer` es covariante en `A` ":

```
trait Producer[+A] {
  def produce: A
}
```

Un parámetro de tipo covariante puede considerarse como un tipo de "salida". Marcar `A` como covariante afirma que el `Producer[X] <: Producer[Y]` siempre que `X <: Y` Por ejemplo, un `Producer[Cat]` es un `Producer[Animal]` válido, ya que todos los gatos producidos también son animales válidos.

Un parámetro de tipo covariante no puede aparecer en posición contravariante (entrada). El siguiente ejemplo no se compilará, ya que estamos afirmando que `Co[Cat] <: Co[Animal]` , pero `Co[Cat]` tiene `def handle(a: Cat): Unit` que no puede manejar ningún `Animal` como lo requiere `Co[Animal]` !

```
trait Co[+A] {
  def produce: A
  def handle(a: A): Unit
}
```

Un enfoque para lidiar con esta restricción es usar parámetros de tipo limitados por el parámetro de tipo covariante. En el siguiente ejemplo, sabemos que `B` es un supertipo de `A` Por lo tanto, dada la `Option[X] <: Option[Y]` para `X <: Y` , sabemos que la `Option[X] def getOrElse[B >: X](b: => B): B` puede aceptar cualquier supertipo de `X` - que incluye los supertipos de `Y` como lo requiere la `Option[Y]` :

```
trait Option[+A] {
  def getOrElse[B >: A](b: => B): B
}
```

Invariancia

Por defecto, todos los parámetros de tipo son invariantes. Dado el `trait A[B]` , decimos que " `A` es invariante en `B` ". Esto significa que, dadas las dos parametrizaciones `A[Cat]` y `A[Animal]` , no afirmamos que exista una relación de subclase / superclase entre estos dos tipos; no sostiene que `A[Cat] <: A[Animal]` ni que `A[Cat] >: A[Animal]` independientemente de la relación entre el `Cat` y el `Animal` .

Las anotaciones de varianza nos proporcionan un medio para declarar tal relación e impone reglas sobre el uso de parámetros de tipo para que la relación siga siendo válida.

Contravarianza

El símbolo `-` marca un parámetro de tipo como *contravariante* - aquí decimos que "El `Handler` es contravariante en `A` ":

```
trait Handler[-A] {
  def handle(a: A): Unit
}
```

Un parámetro de tipo contravariante se puede considerar como un tipo de "entrada". Marcar `A` como contravariante afirma que `Handler[X] <: Handler[Y]` siempre que `X >: Y` Por ejemplo, un `Handler[Animal]` es un `Handler[Cat]` válido `Handler[Cat]` , como un `Handler[Animal]` también debe manejar gatos.

Un parámetro de tipo contravariante no puede aparecer en posición covariante (salida). El siguiente ejemplo no se compilará, ya que estamos afirmando que un `Contra[Animal] <: Contra[Cat]` , sin embargo, un `Contra[Animal]` tiene `def produce: Animal` que no está garantizado que produzca gatos como lo exige `Contra[Cat]` .

```
trait Contra[-A] {
   def handle(a: A): Unit
   def produce: A
}
```

Sin embargo, tenga en cuenta que, a los fines de la resolución de sobrecarga, la contravarianza también invierte de forma contraintuitiva la especificidad de un tipo en el parámetro de tipo contravariante: se considera que " `Handler[Animal]` es" más específico "que `Handler[Cat]` .

Como no es posible sobrecargar los métodos en los parámetros de tipo, este comportamiento generalmente solo se vuelve problemático al resolver argumentos implícitos. En el siguiente ejemplo, nunca se usará `ofCat` , ya que el tipo de retorno de `ofAnimal` es más específico:

```
implicit def ofAnimal: Handler[Animal] = ???
implicit def ofCat: Handler[Cat] = ???

implicitly[Handler[Cat]].handle(new Cat)
```

Este comportamiento está actualmente programado para cambiar en punto y es por eso que (como ejemplo) `scala.math.Ordering` es invariante en su tipo de parámetro `T` Una solución es hacer que su clase de tipo sea invariante, y parametrizar la definición implícita en el caso de que quiera que se aplique a las subclases de un tipo dado:

```
trait Person
object Person {
  implicit def ordering[A <: Person]: Ordering[A] = ???
}
```

Covarianza de una colección.

Debido a que las colecciones son típicamente covariantes en su tipo de elemento *, se puede pasar una colección de un subtipo donde se espera un supertipo:

```
trait Animal { def name: String }
case class Dog(name: String) extends Animal

object Animal {
  def printAnimalNames(animals: Seq[Animal]) = {
    animals.foreach(animal => println(animal.name))
  }
}

val myDogs: Seq[Dog] = Seq(Dog("Curly"), Dog("Larry"), Dog("Moe"))

Animal.printAnimalNames(myDogs)
// Curly
// Larry
// Moe
```

Puede que no parezca magia, pero el hecho de que un `Seq[Dog]` sea aceptado por un método que espera un `Seq[Animal]` es el concepto completo de un tipo de tipo superior (aquí: `Seq`) que es covariante en su parámetro de tipo.

* Un contraejemplo es el conjunto de la biblioteca estándar.

La covarianza en un rasgo invariante

También hay una forma de que un solo método acepte un argumento covariante, en lugar de tener todo el rasgo covariante. Esto puede ser necesario porque le gustaría usar `T` en una posición contravariante, pero aún así es covariante.

```
trait LocalVariance[T]{
  /// ??? throws a NotImplementedError
  def produce: T = ???
  // the implicit evidence provided by the compiler confirms that S is a
  // subtype of T.
  def handle[S](s: S)(implicit evidence: S <:< T) = {
    // and we can use the evidence to convert s into t.
    val t: T = evidence(s)
    ???
  }
}

trait A {}
trait B extends A {}

object Test {
  val lv = new LocalVariance[A] {}

  // now we can pass a B instead of an A.
  lv.handle(new B {})
}
```

Lea Variación de tipo en línea: https://riptutorial.com/es/scala/topic/1651/variacion-de-tipo

Capítulo 62: Zurra

Sintaxis

- aFunction (10) _ // Using '_' Le dice al compilador que todos los parámetros en el resto de los grupos de parámetros serán procesados.
- nArityFunction.curried // Convierte una función n-arity a una versión con currículum equivalente
- anotherFunction (x) (_: String) (z) // Currying un parámetro arbitrario. Necesita su tipo explícitamente establecido.

Examples

Un multiplicador configurable como función de curry.

```
def multiply(factor: Int)(numberToBeMultiplied: Int): Int = factor * numberToBeMultiplied

val multiplyBy3 = multiply(3)_     // resulting function signature Int => Int
val multiplyBy10 = multiply(10)_ // resulting function signature Int => Int

val sixFromCurriedCall = multiplyBy3(2) //6
val sixFromFullCall = multiply(3)(2)    //6

val fortyFromCurriedCall = multiplyBy10(4) //40
val fortyFromFullCall = multiply(10)(4)    //40
```

Múltiples grupos de parámetros de diferentes tipos, parámetros de posiciones arbitrarias

```
def numberOrCharacterSwitch(toggleNumber: Boolean)(number: Int)(character: Char): String =
  if (toggleNumber) number.toString else character.toString

// need to explicitly specify the type of the parameter to be curried
// resulting function signature Boolean => String
val switchBetween3AndE = numberOrCharacterSwitch(_: Boolean)(3)('E')

switchBetween3AndE(true) // "3"
switchBetween3AndE(false) // "E"
```

Currying una función con un solo grupo de parámetros

```
def minus(left: Int, right: Int) = left - right

val numberMinus5 = minus(_: Int, 5)
val fiveMinusNumber = minus(5, _: Int)

numberMinus5(7)    //  2
fiveMinusNumber(7) // -2
```

Zurra

Vamos a definir una función de 2 argumentos:

```
def add: (Int, Int) => Int = (x,y) => x + y
val three = add(1,2)
```

La `add` curry lo transforma en una función que toma **un** `Int` y devuelve una **función** (de **un** `Int` a un `Int`)

```
val addCurried: (Int) => (Int => Int) = add2.curried
//                ^~~ take *one* Int
//                          ^~~~ return a *function* from Int to Int

val add1: Int => Int = addCurried(1)
val three: Int = add1(2)
val allInOneGo: Int = addCurried(1)(2)
```

Puede aplicar este concepto a cualquier función que tome múltiples argumentos. El curry de una función que toma múltiples argumentos, la transforma en una serie de aplicaciones de funciones que toman **un** argumento:

```
def add3: (Int, Int, Int) => Int = (a,b,c) => a + b + c + d
def add3Curr: Int => (Int => (Int => Int)) = add3.curried

val x = add3Curr(1)(2)(42)
```

Zurra

Currying, según Wikipedia ,

> es la técnica de traducir la evaluación de una función que toma múltiples argumentos para evaluar una secuencia de funciones.

Concretamente, en términos de tipos de escala, en el contexto de una función que toma dos argumentos, (tiene aridad 2) es la conversión de

```
val f: (A, B) => C // a function that takes two arguments of type `A` and `B` respectively
                   // and returns a value of type `C`
```

a

```
val curriedF: A => B => C // a function that take an argument of type `A`
                          // and returns *a function*
                          // that takes an argument of type `B` and returns a `C`
```

Así que para las funciones arity-2 podemos escribir la función curry como:

```
def curry[A, B, C](f: (A, B) => C): A => B => C = {
  (a: A) => (b: B) => f(a, b)
```

```
}
```

Uso:

```
val f: (String, Int) => Double = {(_, _) => 1.0}
val curriedF: String => Int => Double = curry(f)
f("a", 1)        // => 1.0
curriedF("a")(1) // => 1.0
```

Scala nos da algunas características de lenguaje que ayudan con esto:

1. Puede escribir funciones al curry como métodos. así que `curriedF` se puede escribir como:

```
def curriedFAsAMethod(str: String)(int: Int): Double = 1.0
val curriedF = curriedFAsAMethod _
```

2. Puede deshacer el curry (es decir, pasar de `A => B => C` a `(A, B) => C`) utilizando un método de biblioteca estándar: `Function.uncurried`

```
val f: (String, Int) => Double = Function.uncurried(curriedF)
f("a", 1) // => 1.0
```

Cuando usar Currying

El curry *es la técnica de traducir la evaluación de una función que toma múltiples argumentos para evaluar una secuencia de funciones, cada una con un solo argumento* .

Esto suele ser útil cuando, por ejemplo:

1. Diferentes argumentos de una función se calculan **en diferentes momentos** . *(Ejemplo 1)*
2. Los diferentes argumentos de una función se calculan **por diferentes niveles de la aplicación** . *(Ejemplo 2)*

Ejemplo 1

Supongamos que el ingreso anual total es una función compuesta por el ingreso y una bonificación:

```
val totalYearlyIncome:(Int,Int) => Int =  (income, bonus) => income + bonus
```

La versión al curry de la función 2-aridad anterior es:

```
val totalYearlyIncomeCurried: Int => Int => Int = totalYearlyIncome.curried
```

Tenga en cuenta en la definición anterior que el tipo también se puede ver / escribir como:

```
Int => (Int => Int)
```

Supongamos que la porción del ingreso anual se conoce de antemano:

```
val partialTotalYearlyIncome: Int => Int = totalYearlyIncomeCurried(10000)
```

Y en algún punto de la línea se conoce la bonificación:

```
partialTotalYearlyIncome(100)
```

Ejemplo 2

Supongamos que la fabricación de automóviles implica la aplicación de ruedas y carrocería de automóviles:

```
val carManufacturing:(String,String) => String = (wheels, body) => wheels + body
```

Estas piezas son aplicadas por diferentes fábricas:

```
class CarWheelsFactory {
  def applyCarWheels(carManufacturing:(String,String) => String): String => String =
          carManufacturing.curried("applied wheels..")
}

class CarBodyFactory {
  def applyCarBody(partialCarWithWheels: String => String): String =
partialCarWithWheels("applied car body..")
}
```

Tenga en cuenta que la `CarWheelsFactory` anterior aplica la función de fabricación del automóvil y solo aplica las ruedas.

El proceso de fabricación del automóvil tomará la siguiente forma:

```
val carWheelsFactory = new CarWheelsFactory()
val carBodyFactory   = new CarBodyFactory()

val carManufacturing:(String,String) => String = (wheels, body) => wheels + body

val partialCarWheelsApplied: String => String  =
carWheelsFactory.applyCarWheels(carManufacturing)
val carCompleted = carBodyFactory.applyCarBody(partialCarWheelsApplied)
```

Un uso del mundo real de Currying.

Lo que tenemos es una lista de tarjetas de crédito y nos gustaría calcular las primas de todas las tarjetas que la compañía de tarjetas de crédito tiene que pagar. Las primas dependen del número total de tarjetas de crédito, por lo que la empresa las ajusta en consecuencia.

Ya tenemos una función que calcula la prima de una sola tarjeta de crédito y tiene en cuenta el total de tarjetas que la compañía ha emitido:

```
case class CreditCard(creditInfo: CreditCardInfo, issuer: Person, account: Account)

object CreditCard {
  def getPremium(totalCards: Int, creditCard: CreditCard): Double = { ... }
}
```

Ahora, un enfoque razonable para este problema sería asignar cada tarjeta de crédito a una prima y reducirla a una suma. Algo como esto:

```
val creditCards: List[CreditCard] = getCreditCards()
val allPremiums = creditCards.map(CreditCard.getPremium).sum //type mismatch; found : (Int,
CreditCard) ⇒ Double required: CreditCard ⇒ ?
```

Sin embargo, al compilador no le va a gustar esto, porque `CreditCard.getPremium` requiere dos parámetros. Aplicación parcial al rescate! Podemos aplicar parcialmente el número total de tarjetas de crédito y usar esa función para asignar las tarjetas de crédito a sus primas. Todo lo que necesitamos hacer es curry la función `getPremium` cambiándola para usar múltiples listas de parámetros y estamos `getPremium` .

El resultado debe verse algo como esto:

```
object CreditCard {
  def getPremium(totalCards: Int)(creditCard: CreditCard): Double = { ... }
}

val creditCards: List[CreditCard] = getCreditCards()

val getPremiumWithTotal = CreditCard.getPremium(creditCards.length)_

val allPremiums = creditCards.map(getPremiumWithTotal).sum
```

Lea Zurra en línea: https://riptutorial.com/es/scala/topic/1636/zurra

Creditos

S. No	Capítulos	Contributors
1	Empezando con Scala Language	4444, Andy Hayden, Ani Menon, Community, David G., David Portabella, dk14, Donald.McLean, Gabriele Petronella, Grzegorz Oledzki, implicitdef, isaias-b, J Atkin, Jean, Jonathan, mammothbane, marcospereira, Marek Skiba, mdarwin, Nathaniel Ford, NeoWelkin, Nicofisi, Priya, rolve, Shoe, sschaef, Thomas Andrews, Tyler James Harden, Ven, Vogon Jeltz
2	Alcance	Camilo Sampedro
3	Anotaciones	Gábor Bakos, Nathaniel Ford, Thomas Matecki
4	Auto tipos	Gábor Bakos, irundaia
5	Biblioteca de continuaciones	dmitry, HTNW
6	Clase de opción	Bruce Lowe, CPS, earldouglas, evan.oman, Governa, John Starich, Matthew Scharley, Nathaniel Ford, R Pieters, ScientiaEtVeritas, suj1th, Tzach Zohar, Vasiliy Levykin
7	Clases de casos	Andy Hayden, Dan Simon, dk14, Gábor Bakos, HTNW, J Cracknell, keegan, made raka teja, Marc Grue, Nathaniel Ford, pedrorijo91, Rumoku, ScientiaEtVeritas, suj1th, Suma
8	Clases de tipo	Arseniy Zhizhelev, Daniel C. Sobral, Gábor Bakos, gregghz, Nathaniel Ford, TomTom, Yawar
9	Clases y objetos	Aamir, Gábor Bakos, mdarwin, mirosval, MSmedberg, Nathaniel Ford, ScientiaEtVeritas, steve, Sudhir Singh, Tzach Zohar, vivek
10	Colecciones	Anton, Camilo Sampedro, deepkimo, Donald.McLean, doub1ejack, EdgeCaseBerg, Filippo Vitale, George, implicitdef, ipoteka, Jason, John Starich, Mr D, Nathaniel Ford, raam86, Shastick, Suma, Tundebabzy, Vasiliy Levykin
11	Colecciones paralelas	Nathaniel Ford, Shuklaswag
12	Combinadores de analizador	Nathaniel Ford
13	Configurando Scala	Hristo Iliev, Matas Vaitkevicius, Nathaniel Ford, Rjk

14	Corrientes	jwvh, Nathaniel Ford, Oleg Pyzhcov
15	Cuasiquotes	gregghz
16	Enumeraciones	Andy Hayden, Cortwave, Daniel Schröter, Gábor Bakos, implicitdef, ipoteka, Nathaniel Ford, phantomastray, Red Mercury
17	Expresiones regulares	dmitry, J Cracknell, Nathaniel Ford
18	Extractores	Andy Hayden, Dan Hulme, Dan Simon, Gábor Bakos, gilad hoch, Idloj, J Cracknell, jwvh, knutwalker, Łukasz, Martin Seeler, Michael Ahlers, Nathaniel Ford, Suma, W.P. McNeill
19	Función de orden superior	acjay, ches, Nathaniel Ford, nukie, Rajat Jain, Srini
20	Funciones	Aravindh S, Archeg, Camilo Sampedro, ches, corvus_192, Dawny33, Gábor Bakos, Gabriele Petronella, implicitdef, ipoteka , Jean, jwvh, michael_s, Nathaniel Ford, raam86, rjsvaljean, ScientiaEtVeritas, Shastick, stefanobaghino, Sven Koschnicke, vise890, wheaties
21	Funciones definidas por el usuario para Hive	Camilo Sampedro
22	Funciones parciales	acjay, Akash Sethi, David Leppik, dimitrisli, jwvh, Suma, Tzach Zohar
23	Futuros	isaias-b, kevin628, Nathaniel Ford, nukie, Shastick
24	Implícitos	Andy Hayden, dimitrisli, Gábor Bakos, HTNW, implicitdef, ipoteka, Jose Antonio Jimenez Saez, Michael Zajac, Nathaniel Ford, nattyddubbs, Simon, spiffman, Suma, Timo, vsminkov
25	Inferencia de tipos	Gábor Bakos, Nathaniel Ford, suj1th
26	Interoperabilidad de Java	Andrzej Jozwik, Dan Hulme, Gábor Bakos, mvn, the21st, thekingofkings
27	Interpolación de cuerdas	Andy Hayden, Ayberk, Brian, implicitdef, J Cracknell, Nadim Bahadoor
28	Invocación Dinámica	HTNW
29	Inyección de dependencia	Hoang Ong

30	JSON	ipoteka, John, Muki, Nathaniel Ford, pedrorijo91, suj1th, void, Wogan, zoitol
31	La coincidencia de patrones	Ali Dehghani, Andrzej Jozwik, Andy Hayden, CPS, Dan Simon, Daniel Werner, Filippo Vitale, Gábor Bakos, implicitdef, insan-e, jilen, jozic, JRomero, Justin Bailey, Louis F., mammothbane, Matt, Nadim Bahadoor, Nathaniel Ford, Peter Neyens, Sergio, Shastick, Shoe, Simon, Suma, T.Grottker, user6062072, vdebergue, vsminkov, Yagüe
32	Macros	gregghz, HTNW, Nathaniel Ford
33	Manejo de errores	Andy Hayden, Graham, John Starich, made raka teja, mnoronha, Nathaniel Ford, Simon, Suma, tacos_tacos_tacos, Tzach Zohar
34	Manejo de XML	Nathaniel Ford, Rockie Yang, vsnyc
35	Mejores prácticas	corvus_192, ipoteka, Nathaniel Ford, RamenChef, Sarvesh Kumar Singh, Shuklaswag
36	Mientras bucles	J Cracknell, Nathaniel Ford
37	Mónadas	ipoteka, Nathaniel Ford
38	Operadores en Scala	Gábor Bakos, Shaido, Suminda Sirinath S. Dharmasena
39	Paquetes	Alex Javarotti, Nathaniel Ford, NetanelRabinowitz
40	Para expresiones	Andy Hayden, J Cracknell, jwvh, LivingRobot, Nathaniel Ford, ScientiaEtVeritas
41	Programación a nivel de tipo	J Cracknell
42	Pruebas con ScalaCheck	Andrzej Jozwik
43	Pruebas con ScalaTest	Nadim Bahadoor, Nathaniel Ford
44	Rasgos	André Laszlo, Andy Hayden, Donald.McLean, Louis F., Nathaniel Ford, Rumoku, Sudhir Singh, Vogon Jeltz
45	Recursion	Dmitry Bystritsky, Gábor Bakos, jilen, jwvh, michael_s, ScientiaEtVeritas, teldosas
46	Reflexión	Sachin Janani
47	Scala.js	Camilo Sampedro

48	Scaladoc	Camilo Sampedro, Gábor Bakos, Nathaniel Ford
49	Scalaz	chengpohi
50	Si expresiones	corvus_192, Nathaniel Ford, ScientiaEtVeritas
51	Símbolos literales	ZbyszekKr
52	sincronizado	Gábor Bakos
53	Sobrecarga del operador	corvus_192, implicitdef, inzi, mnoronha, Nathaniel Ford, Simon
54	Tipo de parametrización (genéricos)	akauppi, Andy Hayden, Eero Helenius, Nathaniel Ford, vivek
55	Tipos de métodos abstractos únicos (tipos SAM)	Gábor Bakos, Gabriele Petronella, Nathaniel Ford
56	Trabajando con datos en estilo inmutable.	Filippo Vitale
57	Trabajando con gradle	Bianca Tesila, Nathaniel Ford, Rjk
58	Tuplas	corvus_192, evan.oman, Lawsy, Nathaniel Ford
59	Unidades de manipulación (medidas)	Gábor Bakos
60	Var, Val y Def	Aamir, John Starich, jwvh, linkhyrule5, Nathaniel Ford, Shastick, Shuklaswag, stefanobaghino, ZbyszekKr
61	Variación de tipo	acjay, J Cracknell, Reactormonk
62	Zurra	Adamos Loizou, alphaloop, Amr Gawish, dimitrisli, Luka Jacobowitz, Nathaniel Ford, rjsvaljean, Suma, vise890